CHAPTER 7

MongoDB Architecture

"MongoDB architecture covers the deep-dive architectural concepts of MongoDB."

In this chapter, you will learn about the MongoDB architecture, especially core processes and tools, standalone deployment, sharding concepts, replication concepts, and production deployment.

Core Processes

The core components in the MongoDB package are

- **mongod**, which is the core database process
- **mongos**, which is the controller and query router for sharded clusters
- **mongo**, which is the interactive MongoDB shell

These components are available as applications under the bin folder. Let's discuss these components in detail.

mongod

The primary daemon in a MongoDB system is known as mongod. This daemon handles all the data requests, manages the data format, and performs operations for background management.

When a mongod is run without any arguments, it connects to the default data directory, which is `C:\data\db` or `/data/db`, and default port 27017, where it listens for socket connections.

It's important to ensure that the data directory exists and you have write permissions to the directory before the mongod process is started.

If the directory doesn't exist or you don't have write permissions on the directory, the start of this process will fail. If the default port 27017 is not available, the server will fail to start.

mongod also has a HTTP server which listens on a port 1000 higher than the default port, so if you started the mongod with the default port 27017, in this case the HTTP server will be on port 28017 and will be accessible using the URL `http://localhost:28017`. This basic HTTP server provides administrative information about the database.

S.G. Edward and N. Sabharwal, *Practical MongoDB*, DOI 10.1007/978-1-4842-0647-8_7

mongo

mongo provides an interactive JavaScript interface for the developer to test queries and operations directly on the database and for the system administrators to manage the database. This is all done via the command line. When the mongo shell is started, it will connect to the default database called `test`. This database connection value is assigned to global variable `db`.

As a developer or administrator you need to change the database from `test` to your database post the first connection is made. You can do this by using `<databasename>.`

mongos

mongos is used in MongoDB sharding. It acts as a routing service that processes queries from the application layer and determines where in the sharded cluster the requested data is located.

We will discuss mongos in more detail in the sharding section. Right now you can think of mongos as the process that routes the queries to the correct server holding the data.

MongoDB Tools

Apart from the core services, there are various tools that are available as part of the MongoDB installation:

- **mongodump**: This utility is used as part of an effective backup strategy. It creates a binary export of the database contents.
- **mongorestore**: The binary database dump created by the mongodump utility is imported to a new or an existing database using the mongorestore utility.
- **bsondump**: This utility converts the BSON files into human-readable formats such as JSON and CSV. For example, this utility can be used to read the output file generated by mongodump.
- **mongoimport**, **mongoexport**: mongoimport provides a method for taking data in JSON, CSV, or TSV formats and importing it into a mongod instance. mongoexport provides a method to export data from a mongod instance into JSON, CSV, or TSV formats.
- **mongostat**, **mongotop**, **mongosniff**: These utilities provide diagnostic information related to the current operation of a mongod instance.

Standalone Deployment

Standalone deployment is used for development purpose; it doesn't ensure any redundancy of data and it doesn't ensure recovery in case of failures. So it's not recommended for use in production environment. Standalone deployment has the following components: a single mongod and a client connecting to the mongod, as shown in Figure 7-1.

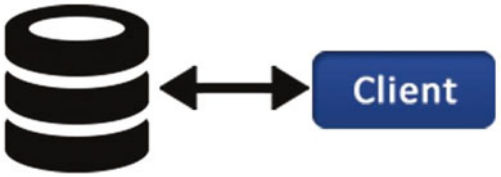


Figure 7-1. *Standalone deployment*

MongoDB uses sharding and replication to provide a highly available system by distributing and duplicating the data. In the coming sections, you will look at sharding and replication. Following that you'll look at the recommended production deployment architecture.

Replication

In a standalone deployment, if the mongod is not available, you risk losing all the data, which is not acceptable in a production environment. Replication is used to offer safety against such kind of data loss.

Replication provides for data redundancy by replicating data on different nodes, thereby providing protection of data in case of node failure. Replication provides high availability in a MongoDB deployment.

Replication also simplifies certain administrative tasks where the routine tasks such as backups can be offloaded to the replica copies, freeing the main copy to handle the important application requests.

In some scenarios, it can also help in scaling the reads by enabling the client to read from the different copies of data.

In this section, you will learn how replication works in MongoDB and its various components. There are two types of replication supported in MongoDB: traditional master/slave replication and replica set.

Master/Slave Replication

In MongoDB, the traditional master/slave replication is available but it is recommended only for more than 50 node replications. The preferred replication approach is replica sets, which we will explain later. In this type of replication, there is one master and a number of slaves that replicate the data from the master. The only advantage with this type of replication is that there's no restriction on the number of slaves within a cluster. However, thousands of slaves will overburden the master node, so in practical scenarios it's better to have less than dozen slaves. In addition, this type of replication doesn't automate failover and provides less redundancy.

In a basic master/slave setup, you have two types of mongod instances: one instance is in the master mode and the remaining are in the slave mode, as shown in Figure 7-2. Since the slaves are replicating from the master, all slaves need to be aware of the master's address.

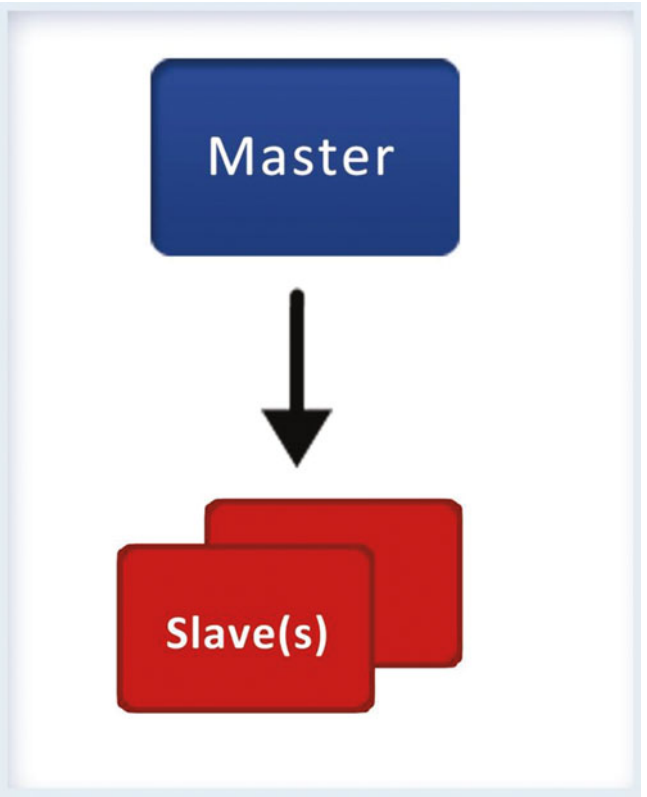


Figure 7-2. *Master/slave replication*

The master node maintains a capped collection (oplog) that stores an ordered history of logical writes to the database.

The slaves replicate the data using this oplog collection. Since the oplog is a capped collection, if the slave's state is far behind the master's state, the slave may become out of sync. In that scenario, the replication will stop and manual intervention will be needed to re-establish the replication.

There are two main reasons behind a slave becoming out of sync:

- The slave shuts down or stops and restarts later. During this time, the oplog may have deleted the log of operations required to be applied on the slave.
- The slave is slow in executing the updates that are available from the master.

Replica Set

The replica set is a sophisticated form of the traditional master-slave replication and is a recommended method in MongoDB deployments.

Replica sets are basically a type of master-slave replication but they provide automatic failover. A replica set has one master, which is termed as primary, and multiple slaves, which are termed as secondary in the replica set context; however, unlike master-slave replication, there's no one node that is fixed to be primary in the replica set.

If a master goes down in replica set, automatically one of the slave nodes is promoted to the master. The clients start connecting to the new master, and both data and application will remain available. In a replica set, this failover happens in an automated fashion. We will explain the details of how this process happens later.

The primary node is selected through an election mechanism. If the primary goes down, the selected node will be chosen as the primary node.

Figure 7-3 shows how a two-member replica set failover happens. Let's discuss the various steps that happen for a two-member replica set in failover.

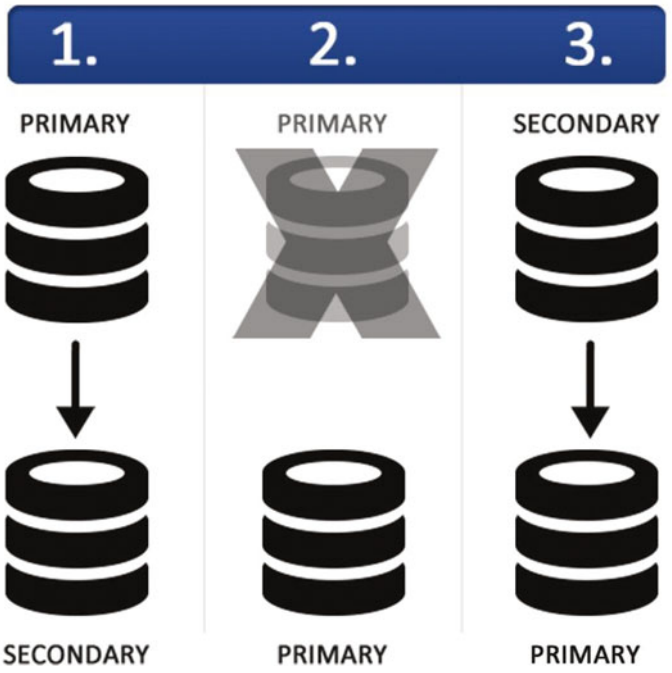


Figure 7-3. *Two-member replica set failover*

1. The primary goes down, and the secondary is promoted as primary.
2. The original primary comes up, it acts as slave, and becomes the secondary node.

The points to be noted are

- A replica set is a mongod's cluster, which replicates among one another and ensures automatic failover.
- In the replica set, one mongod will be the primary member and the others will be secondary members.
- The primary member is elected by the members of the replica set. All writes are directed to the primary member whereas the secondary members replicate from the primary asynchronously using oplog.
- The secondary's data sets reflect the primary data sets, enabling them to be promoted to primary in case of unavailability of the current primary.

Replica set replication has a limitation on the number of members. Prior to version 3.0, the limit was 12 but this has been changed to 50 in version 3.0. So now replica set replication can have maximum of 50 members only, and at any given point of time in a 50-member replica set, only 7 can participate in a vote. We will explain the voting concept in a replica set in detail.

Starting from Version 3.0, replica set members can use different storage engines. For example, the WiredTiger storage engine might be used by the secondary members whereas the MMAPv1 engine could be used by the primary. In the coming sections, you will look at the different storage engines available with MongoDB.

Primary and Secondary Members

Before you move ahead and look at how the replica set functions, let's look at the type of members that a replica set can have. There are two types of members: primary members and secondary members.

- **Primary member**: A replica set can have only one primary, which is elected by the voting nodes in the replica set. Any node with associated priority as 1 can be elected as a primary. The client redirects all the write operations to the primary member, which is then later replicated to the secondary members.
- **Secondary member:** A normal secondary member holds the copy of the data. The secondary member can vote and also can be a candidate for being promoted to primary in case of failover of the current primary.

In addition to this, a replica set can have other types of secondary members.

Types of Secondary Members

Priority 0 members are secondary members that maintain the primary's data copy but can never become a primary in case of a failover. Apart from that, they function as a normal secondary node, and they can participate in voting and can accept read requests. The Priority 0 members are created by setting the priority to 0.

Such types of members are specifically useful for the following reasons:

1. They can serve as a cold standby.
2. In replica sets with varied hardware or geographic distribution, this configuration ensures that only the qualified members get elected as primary.
3. In a replica set that spans multiple data centers across network partitioning, this configuration can help ensure that the main data center has the eligible primary. This is used to ensure that the failover is quick.

Hidden members are 0-priority members that are hidden from the client applications. Like the 0-priority members, this member also maintains a copy of the primary's data, cannot become the primary, and can participate in the voting, but unlike 0-priotiy members, it can't serve any read requests or receive any traffic beyond what replication requires. A node can be set as hidden member by setting the hidden property to true. In a replica set, these members can be dedicated for reporting needs or backups.

Delayed members are secondary members that replicate data with a delay from the primary's oplog. This helps to recover from human errors, such as accidentally dropped databases or errors that were caused by unsuccessful application upgrades.

When deciding on the delay time, consider your maintenance period and the size of the oplog. The delay time should be either equal to or greater than the maintenance window and the oplog size should be set in a manner to ensure that no operations are lost while replicating.

Note that since the delayed members will not have up-to-date data as the primary node, the priority should be set to 0 so that they cannot become primary. Also, the hidden property should be true in order to avoid any read requests.

Arbiters are secondary members that do not hold a copy of the primary's data, so they can never become the primary. They are solely used as member for participating in voting. This enables the replica set to have an uneven number of nodes without incurring any replication cost which arises with data replication.

Non-voting members hold the primary's data copy, they can accept client read operations, and they can also become the primary, but they cannot vote in an election.

The voting ability of a member can be disabled by setting its votes to 0. By default every member has one vote. Say you have a replica set with seven members. Using the following commands in mongo shell, the votes for fourth, fifth, and sixth member are set to 0:

```
cfg_1 = rs.conf()
cfg_1.members[3].votes = 0
cfg_1.members[4].votes = 0
cfg_1.members[5].votes = 0
rs.reconfig(cfg_1)
```

Although this setting allows the fourth, fifth, and sixth members to be elected as primary, when voting their votes will not be counted. They become non-voting members, which means they can stand for election but cannot vote themselves.

You will see how the members can be configured later in this chapter.

Elections

In this section, you will look at the process of election for selecting a primary member. In order to get elected, a server need to not just have the majority but needs to have majority of the total votes.

If there are X servers with each server having 1 vote, then a server can become primary only when it has at least [(X/2) + 1] votes.

If a server gets the required number of votes or more, then it will become primary.

The primary that went down still remains part of the set; when it is up, it will act as a secondary server until the time it gets a majority of votes again.

The complication with this type of voting system is that you cannot have just two nodes acting as master and slave. In this scenario, you will have total of two votes, and to become a master, a node will need the majority of votes, which will be both of the votes in this case. If one of the servers goes down, the other server will end up having one vote out of two, and it will never be promoted as master, so it will remain a slave.

In case of network partitioning, the master will lose the majority of votes since it will have only its own one vote and it'll be demoted to slave and the node that is acting as slave will also remain a slave in the absence of the majority of the votes. You will end up having two slaves until both servers reach each other again.

A replica set has number of ways to avoid such situations. The simplest way is to use an arbiter to help resolve such conflicts. It's very lightweight and is just a voter, so it can run on either of the servers itself.

Let's now see how the above scenario will change with the use of an arbiter. Let's first consider the network partitioning scenario. If you have a master, a slave, and an arbiter, each has one vote, totalling three votes. If a network partition occurs with the master and arbiter in one data center and the slave in another data center, the master will remain master since it will still have the majority of votes.

If the master fails with no network partitioning, the slave can be promoted to master because it will have two votes (slave + arbiter).

This three-server setup provides a robust failover deployment.

Example - Working of Election Process in More Details

This section will explain how the election happens.

Let's assume you have a replica set with the following three members: A1, B1, and C1. Each member exchanges a heartbeat request with the other members every few seconds. The members respond with their current situation information to such requests. A1 sends out heartbeat request to B1 and C1. B1 and C1 respond with their current situation information, such as the state they are in (primary or secondary), their current clock time, their eligibility to be promoted as primary, and so on. A1 receives all this information's and updates its "map" of the set, which maintains information such as the members changed state, members that have gone down or come up, and the round trip time.

While updating the A1's map changes, it will check a few things depending on its state:

- If A1 is primary and one of the members has gone down, then it will ensure that it's still able to reach the majority of the set. If it's not able to do so, it will demote itself to secondary state.

  Demotions: There's a problem when A1 undergoes a demotion. By default in MongoDB writes are fire-and-forget (i.e. the client issues the writes but doesn't wait for a response). If an application is doing the default writes when the primary is stepping down, it will never realize that the writes are actually not happening and might end up losing data. Hence it's recommended to use safe writes. In this scenario, when the primary is stepping down, it closes all its client connections, which will result in socket errors to the clients. The client libraries then need to recheck who the new primary is and will be saved from losing their write operations data.

- If A1 is a secondary and if the map has not changed, it will occasionally check whether it should elect itself.

  The first task A1 will do is run a sanity check where it will check answers to few question such as, Does A1 think it's already primary? Does another member think its primary? Is A1 not eligible for election? If it can't answer any of the basic questions, A1 will continue idling as is; otherwise, it will proceed with the election process:

  - A1 sends a message to the other members of the set, which in this case are B1 and C1, saying "I am planning to become a primary. Please suggest"
  - When B1 and C1 receive the message, they will check the view around them. They will run through a big list of sanity checks, such as, Is there any other node that can be primary? Does A1 have the most recent data or is there any other node that has the most recent data? If all the checks seem ok, they send a "go-ahead" message; however, if any of the checks fail, a "stop election" message is sent.
  - If any of the members send a "stop election" reply, the election is cancelled and A1 remains a secondary member.
  - If the "go-ahead" is received from all, A1 goes to the election process final phase.

In the second (final) phase,

- A1 resends a message declaring its candidacy for the election to the remaining members.
- Members B1 and C1 do a final check to ensure that all the answers still hold true as before.
- If yes, A1 is allowed to take its election lock, which prevents its voting capabilities for 30 seconds and sends back a vote.
- If any of the checks fail to hold true, a veto is sent.
- If any veto is received, the election stops.
- If no one vetoes and A1 gets a majority of the votes, it becomes a primary.

The election is affected by the priority settings. A 0 priority member can never become a primary.

Data Replication Process

Let's look at how the data replication works. The members of a replica set replicate data continuously. Every member, including the primary member, maintains an oplog. An oplog is a capped collection where the members maintain a record of all the operations that are performed on the data set.

The secondary members copy the primary member's oplog and apply all the operations in an asynchronous manner.

Oplog

Oplog stands for the operation log. An oplog is a capped collection where all the operations that modify the data are recorded.

The oplog is maintained in a special database, namely `local` in the collection `oplog.$main`. Every operation is maintained as a document, where each document corresponds to one operation that is performed on the master server. The document contains various keys, including the following keys:

- `ts`: This stores the timestamp when the operations are performed. It's an internal type and is composed of a 4-byte timestamp and a 4-byte incrementing counter.
- `op`: This stores information about the type of operation performed. The value is stored as 1-byte code (e.g. it will store an "I" for an insert operation).
- `ns`: This key stores the collection namespace on which the operation was performed.
- `o`: This key specifies the operation that is performed. In case of an insert, this will store the document to insert.

Only operations that change the data are maintained in the oplog because it's a mechanism for ensuring that the secondary node data is in sync with the primary node data.

The operations that are stored in the oplog are transformed so that they remain idempotent, which means that even if it's applied multiple times on the secondary, the secondary node data will remain consistent. Since the oplog is a capped collection, with every new addition of an operation, the oldest operations are automatically moved out. This is done to ensure that it does not grow beyond a pre-set bound, which is the oplog size.

Depending on the OS, whenever the replica set member first starts up, the oplog is created of a default size by MongoDB.

By default in MongoDB, available free space or 5% is used for the oplog on Windows and 64-bit Linux instances. If the size is lower than 1GB, then 1GB of space is allocated by MongoDB.

Although the default size is sufficient in most cases, you can use the `–oplogsize` option to specify the oplog size in MB when starting the server.

If you have the following workload, there might be a requirement of reconsidering the oplog size:

- **Updates to multiple documents simultaneously:** Since the operations need to be translated into operations that are idempotent, this scenario might end up requiring great deal of oplog size.
- **Deletes and insertions happening at the same rate involving same amount of data:** In this scenario, although the database size will not increase, the operations translation into an idempotent operation can lead to a bigger oplog.
- **Large number of in-place updates:** Although these updates will not change the database size, the recording of updates as idempotent operations in the oplog can lead to a bigger oplog.

Initial Sync and Replication

Initial sync is done when the member is in either of the following two cases:

1. The node has started for the first time (i.e. it's a new node and has no data).
2. The node has become stale, where the primary has overwritten the oplog and the node has not replicated the data. In this case, the data will be removed.

In both cases, the initial sync involves the following steps:

1. First, all databases are cloned.
2. Using oplog of the source node, the changes are applied to its dataset.
3. Finally, the indexes are built on all the collections.

Post the initial sync, the replica set members continuously replicate the changes in order to be up-to-date.

Most of the synchronization happens from the primary, but chained replication can be enabled where the sync happens from a secondary only (i.e. the sync targets are changed based on the ping time and state of other member's replication).

Syncing – Normal Operation

In normal operations, the secondary chooses a member from where it will sync its data, and then the operations are pulled from the chosen source's oplog collection (`local.oplog.rs`).

Once the operation (op) is get, the secondary does the following:

1. It first applies the op to its data copy.
2. Then it writes the op to its local oplog.
3. Once the op is written to the oplog, it requests the next op.

Suppose it crashes between step 1 and step 2, and then it comes back again. In this scenario, it'll assume the operation has not been performed and will re-apply it.

Since oplog ops are idempotent, the same operation can be applied any number of times, and every time the result document will be same.

If you have the following doc

{I:11}

an increment operation is performed on the same, such as

{$inc:{I:1}} on the primary

In this case the following will be stored in the primary oplog:

{I:12}.

This will be replicated by the secondaries. So the value remains the same even if the log is applied multiple times.

Starting Up

When a node is started, it checks its local collection to find out the `lastOpTimeWritten`. This is the time of the latest op that was applied on the secondary.

The following shell helper can be used to find the latest op in the shell:

```
> rs.debug.getLastOpWritten()
```

The output returns a field named `ts`, which depicts the last op time.

If a member starts up and finds the `ts` entry, it starts by choosing a target to sync from and it will start syncing as in a normal operation. However, if no entry is found, the node will begin the initial sync process.

Whom to Sync From?

In this section, you will look at how the source is chosen to sync from. As of 2.0, based on the average ping time servers automatically sync from the "nearest" node.

When you bring up a new node, it sends heartbeats to all nodes and monitors the response time. Based on the data received, it then decides the member to sync from using the following algorithm:

```
for each healthy member Loop:
    if state is Primary
        add the member to possible sync target set
    if member's lastOpTimeWritten is greater then the local lastOpTime Written
add the member to possible sync target set
Set sync_from  = MIN (PING TIME to members of sync target set)

Note: A "healthy member" can be thought of as a "normal" primary or secondary member.
```

In version 2.0, the slave's delayed nodes were debatably included in "healthy" nodes. Starting from version 2.2, delayed nodes and hidden nodes are excluded from the "healthy" nodes.

Running the following command will show the server that is chosen as the source for syncing:

```
db.adminCommand({replSetGetStatus:1})
```

The output field of `syncingTo` is present only on secondary nodes and provides information on the node from which it is syncing.

Making Writes Work with Chaining Slaves

You have seen that the above algorithm for choosing a source to sync from implies that slave chaining is semi-automatic. When a server is started, it'll most probably choose a server within the same data center to sync from, thus reducing the WAN traffic.

However, this will never lead to a loop because the nodes will sync only from a secondary that has a latest value of `lastOpTimeWritten` which is greater than its own. You will never end up in a scenario where N1 is syncing from N2 *and* N2 is syncing from N1. It will always be either N1 is syncing from N2 *or* N2 is syncing from N1.

In this section, you will see how w (write operation) works with slave chaining. If N1 is syncing from N2, which is further syncing from N3, in this case how N3 will know that until which point N1 is synced to.

When N1 starts its sync from N2, a special "handshake" message is sent, which intimates to N2 that N1 will be syncing from its oplog. Since N2 is not primary, it will forward the message to the node it is syncing from (i.e. it opens a connection to N3 pretending to be N1). By the end of the above step, N2 has two connections that are opened with N3: one connection for itself and the other for N1.

Whenever an op request is made by N1 to N2, the op is sent by N2 from its oplog and a dummy request is forwarded on the link of N1 to N3, as shown in Figure 7-4.

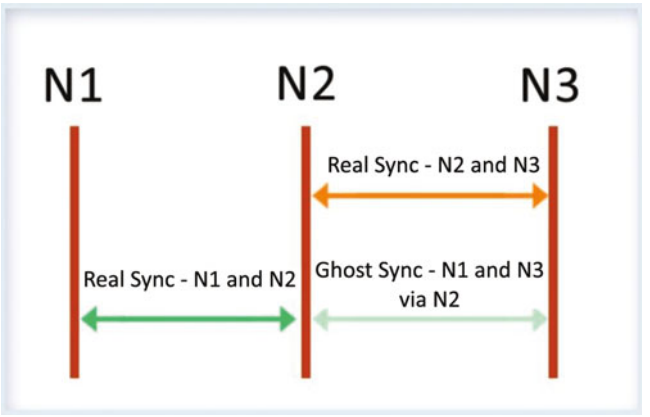


Figure 7-4. *Writes via chaining slaves*

Although this minimizes network traffic, it increases the absolute time for the write to reach to all of the members.

Failover

In this section, you will look at how primary and secondary member failovers are handled in replica sets. All members of a replica set are connected to each other. As shown in Figure 7-5, they exchange a heartbeat message amongst each other.

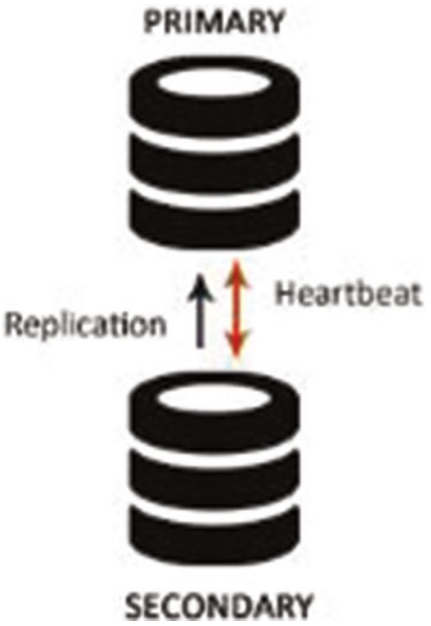


Figure 7-5. *Heartbeat message exchange*

Hence a node with missing heartbeat is considered as crashed.

If the Node Is a Secondary Node

If the node is a secondary node, it will be removed from the membership of the replica set. In the future, when it recovers, it can re-join. Once it re-joins, it needs to update the latest changes.

1. If the down period is small, it connects to the primary and catches up with the latest updates.
2. However, if the down period is lengthy, the secondary server will need to resync with primary where it deletes all its data and does an initial sync as if it's a new server.

If the Node Is the Primary Node

If the node is a primary node, in this scenario if the majority of the members of the original replica sets are able to connect to each other, a new primary will be elected by these nodes, which is in accordance with the automatic failover capability of the replica set.

The election process will be initiated by any node that cannot reach the primary.

The new primary is elected by majority of the replica set nodes. Arbiters can be used to break ties in scenarios such as when network partitioning splits the participating nodes into two halves and the majority cannot be reached.

The node with the highest priority will be the new primary. If you have more than one node with same priority, the data freshness can be used for breaking ties.

The primary node uses a heartbeat to track how many nodes are visible to it. If the number of visible nodes falls below the majority, the primary automatically falls back to the secondary state. This scenario prevents the primary from functioning when it's separated by a network partition.

Rollbacks

In scenario of a primary node change, the data on the new primary is assumed to be the latest data in the system. When the former primary joins back, any operation that is applied on it will also be rolled back. Then it will be synced with the new primary.

The rollback operation reverts all the write operations that were not replicated across the replica set. This is done in order to maintain database consistency across the replica set.

When connecting to the new primary, all nodes go through a resync process to ensure the rollback is accomplished. The nodes look through the operation that is not there on the new primary, and then they query the new primary to return an updated copy of the documents that were affected by the operations. The nodes are in the process of resyncing and are said to be recovering; until the process is complete, they will not be eligible for primary election.

This happens very rarely, and if it happens, it is often due to network partition with replication lag where the secondaries cannot keep up with the operation's throughput on the former primary.

It needs to be noted that if the write operations replicate to other members before the primary steps down, and those members are accessible to majority of the nodes of the replica set, the rollback does not occur.

The rollback data is written to a BSON file with filenames such as `<database>.<collection>.<timestamp>.bson` in the database's `dbpath` directory.

The administrator can decide to either ignore or apply the rollback data. Applying the rollback data can only begin when all the nodes are in sync with the new primary and have rolled back to a consistent state.

The content of the rollback files can be read using `Bsondump`, which then need to be manually applied to the new primary using mongorestore.

There is no method to handle rollback situations automatically for MongoDB. Therefore manual intervention is required to apply rollback data. While applying the rollback, it's vital to ensure that these are replicated to either all or at least some of the members in the set so that in case of any failover rollbacks can be avoided.

Consistency

You have seen that the replica set members keep on replicating data among each other by reading the oplog. How is the consistency of data maintained? In this section, you will look at how MongoDB ensures that you always access consistent data.

In MongoDB, although the reads can be routed to the secondaries, the writes are always routed to the primary, eradicating the scenario where two nodes are simultaneously trying to update the same data set. The data set on the primary node is always consistent.

If the read requests are routed to the primary node, it will always see the up-to-date changes, which means the read operations are always consistent with the last write operations.

However, if the application has changed the read preference to read from secondaries, there might be a probability of user not seeing the latest changes or seeing previous states. This is because the writes are replicated asynchronously on the secondaries.

This behavior is characterized as eventual consistency, which means that although the secondary's state is not consistent with the primary node state, it will eventually become consistent over time.

There is no way that reads from the secondary can be guaranteed to be consistent, except by issuing write concerns to ensure that writes succeed on all members before the operation is actually marked successful. We will be discussing write concerns in a while.

Possible Replication Deployment

The architecture you chose to deploy a replica set affects its capability and capacity. In this section, you will look at few strategies that you need to be aware of while deciding on the architecture. We will also be discussing the deployment architecture.

1. **Odd number of members**: This should be done in order to ensure that there is no tie when electing a primary. If the number of nodes is even, then an arbiter can be used to ensure that the total nodes participating in election is odd, as shown in Figure 7-6.

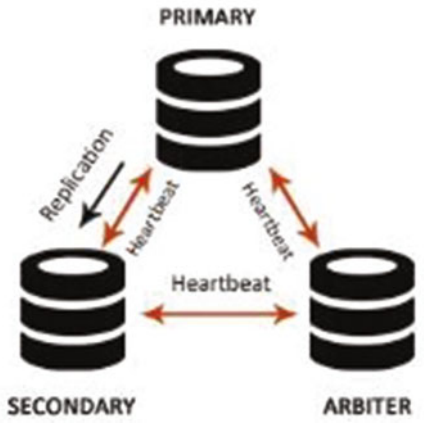


Figure 7-6. *Members replica set with primary, secondary, and arbiter*

2. **Replica set fault tolerance** is the count of members, which can go down but still the replica set has enough members to elect a primary in case of any failure. Table 7-1 indicates the relationship between the member count in the replica set and its fault tolerance. Fault tolerance should be considered when deciding on the number of members.

Table 7-1. *Replica Set Fault Tolerance*

Number of Members	Majority Required for Electing a Primary	Fault Tolerance
3	2	1
4	3	1
5	3	2
6	4	2

3. If the application has **specific dedicated requirements**, such as for reporting or backups, then delayed or hidden members can be considered as part of the replica set, as shown in Figure 7-7.

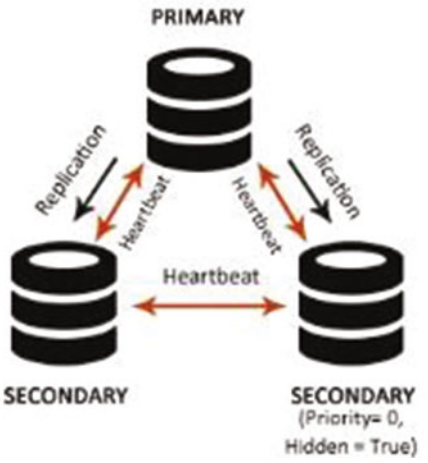


Figure 7-7. *Members replica set with primary, secondary, and hidden members*

4. If the **application is read-heavy**, the read can be distributed across secondaries. As the requirement increases, more nodes can be added to increase the data duplication; this can have a positive impact on the read throughput.

5. The **members should be distributed geographically** in order to cater to main data center failure. As shown in Figure 7-8, the members that are kept at a geographically different location other than the main data center can have priority set as 0, so that they cannot be elected as primary and can act as a standby only.

Figure 7-8. *Members replica set with primary, secondary, and a priority 0 member distributed across the data center*

6. When **replica set members are distributed** across data centers, network partitioning can prevent data centers from communicating with each other. In order to ensure a majority in the case of network partitioning, it keeps a majority of the members in one location.

Scaling Reads

Although the primary purpose of the secondaries is to ensure data availability in case of downtime of the primary node, there are other valid use cases for secondaries. They can be used dedicatedly to perform backup operations or data processing jobs or to scale out reads. One of the ways to scale reads is to issue the read queries against the secondary nodes; by doing so the workload on the master is reduced.

One important point that you need to consider when using secondaries for scaling read operations is that in MongoDB the replication is asynchronous, which means if any write or update operation is performed on the master's data, the secondary data will be momentarily out-of-date. If the application in question is read-heavy and is accessed over a network and does not need up-to-date data, the secondaries can be used to scale out the read in order to provide a good read throughput. Although by default the read requests are routed to the primary node, the requests can be distributed over secondary nodes by specifying the **read preferences**. Figure 7-9 depicts the default read preference.

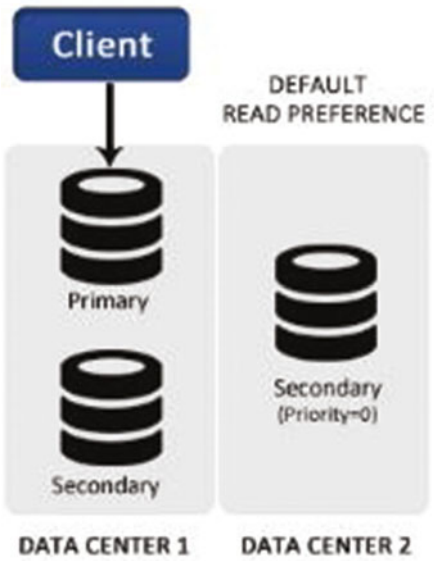


Figure 7-9. *Default read preference*

The following are ideal use cases whereby routing the reads on secondary node can help gain a significant improvement in the read throughput and can also help reduce the latency:

1. **Applications that are geographically distributed**: In such cases, you can have a replica set that is distributed across geographies. The read preferences should be set to read from the **nearest** secondary node. This helps in reducing the latency that is caused when reading over network and this improves the read performance. See Figure 7-10.

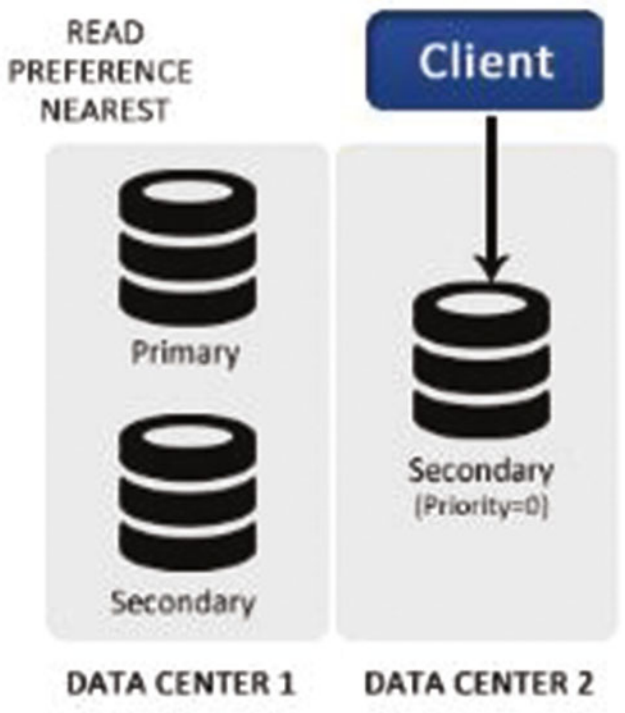


Figure 7-10. *Read Preference – Nearest*

2. If the application always requires up-to-date data, it uses the option `primaryPreferred`, which in normal circumstances will always read from the primary node, but in case of emergency will route the read to secondaries. This is useful during failovers. See Figure 7-11.

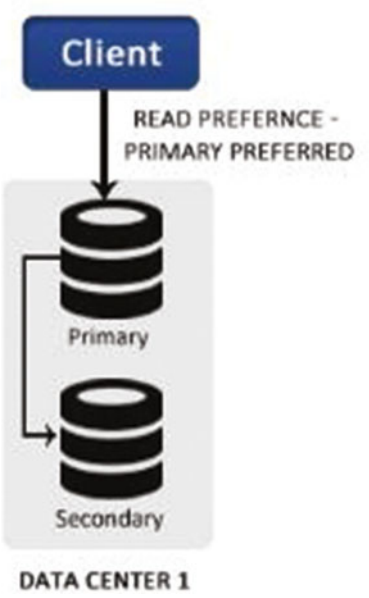


Figure 7-11. *Read Preference – primaryPreferred*

3. If you have an application that supports two types of operations, the first operation is the main workload that involves reading and doing some processing on the data, whereas the second operation generates reports using the data. In such a scenario, you can have the reporting reads directed to the secondaries.

MongoDB supports the following read preference modes:

- **primary**: This is the default mode. All the read requests are routed to the primary node.
- **primaryPreferred**: In normal circumstances the reads will be from primary but in an emergency such as a primary not available, reads will be from the secondary nodes.
- **secondary**: Reads from the secondary members.
- **secondaryPreferred**: Reads from secondary members. If secondaries are unavailable, then read from the primary.
- **nearest**: Reads from the nearest replica set member.

In addition to scaling reads, the second ideal use case for using secondaries is to offload intensive processing, aggregating, and administration tasks in order to avoid degrading the primary's performance. Blocking operations can be performed on the secondary without ever affecting the primary node's performance.

Application Write Concerns

When the client application interacts with MongoDB, it is generally not aware whether the database is on standalone deployment or is deployed as a replica set. However, when dealing with replica sets, the client should be aware of **write concern** and **read concern**.

Since a replica set duplicates the data and stores it across multiple nodes, these two concerns give a client application the flexibility to enforce data consistency across nodes while performing read or write operations.

Using a write concern enables the application to get a success or failure response from MongoDB.

When used in a replica set deployment of MongoDB, the write concern sends a confirmation from the server to the application that the write has succeeded on the primary node. However, this can be configured so that the write concern returns success only when the write is replicated to all the nodes maintaining the data.

In practical scenario, this isn't feasible because it will reduce the write performance. Ideally the client can ensure, using a write concern, that the data is replicated to one more node in addition to the primary, so that the data is not lost even if the primary steps down.

The write concern returns an object that indicates either error or no error.

The `w` option ensures that the write has been replicated to the specified number of members. Either a number or a majority can be specified as the value of the `w` option.

If a number is specified, the write replicates to that many number of nodes before returning success. If a majority is specified, the write is replicated to a majority of members before returning the result.

Figure 7-12 shows how a write happens with w: 2.

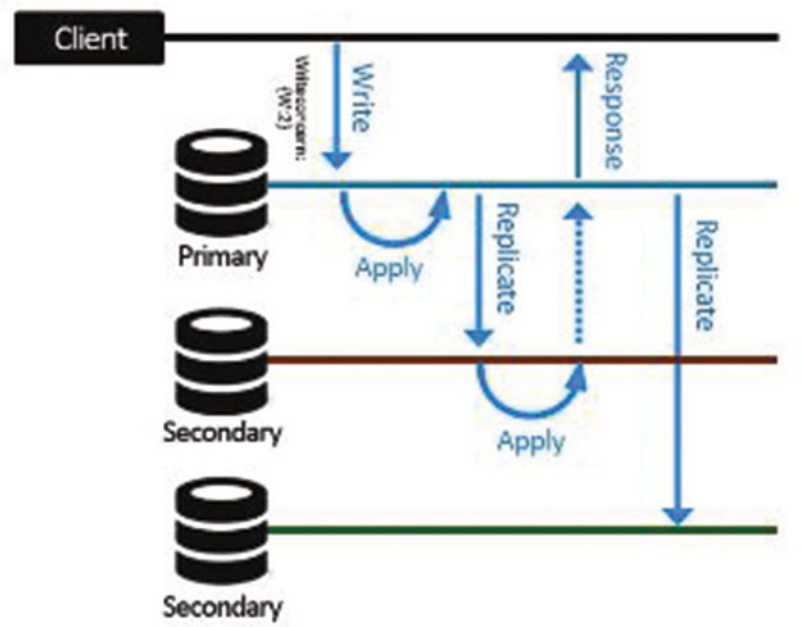


Figure 7-12. *writeConcern*

If while specifying number the number is greater than the nodes that actually hold the data, the command will keep on waiting until the members are available. In order to avoid this indefinite wait time, `wtimeout` should also be used along with `w`, which will ensure that it will wait for the specified time period, and if the write has not succeeded by that time, it will time out.

How Writes Happen with Write Concern

In order to ensure that the written data is present on say at least two members, issue the following command:

```
>db.testprod.insert({i:"test", q: 50, t: "B"}, {writeConcern: {w:2}})
```

In order to understand how this command will be executed, say you have two members, one named primary and the other named secondary, and it is syncing its data from the primary.

But how will the primary know the point at which the secondary is synced? Since the primary's oplog is queried by the secondary for op results to be applied, if the secondary requests an op written at say **t** time, it implies to the primary that the secondary has replicated all ops written before **t**.

The following are the steps that a write concern takes.

1. The write operation is directed to the primary.
2. The operation is written to the oplog of primary with `ts` depicting the time of operation.
3. A `w: 2` is issued, so the write operation needs to be written to one more server before it's marked successful.
4. The secondary queries the primary's oplog for the op, and it applies the op.
5. Next, the secondary sends a request to the primary requesting for ops with `ts` greater than t.
6. At this point, the primary sends an update that the operation until t has been applied by the secondary as it's requesting for ops with `{ts: {$gt: t}}`.
7. The writeConcern finds that a write has occurred on both the primary and secondary, satisfying the `w: 2` criteria, and the command returns success.

Implementing Advanced Clustering with Replica Sets

Having learned the architecture and inner workings of replica sets, you will now focus on administration and usage of replica sets. You will be focusing on the following:

1. Setting up a replica set.
2. Removing a server.
3. Adding a server.
4. Adding an arbiter.
5. Inspecting the status.
6. Forcing a new election of a primary.
7. Using the web interface to inspect the status of the replica set.

The following examples assume a replica set named testset that has the configuration shown in Table 7-2.

Table 7-2. *Replica Set Configuration*

Member	Daemon	Host:Port	Data File Path
Active_Member_1	Mongod	[hostname]:27021	C:\db1\active1\data
Active_Member_2	Mongod	[hostname]:27022	C:\db1\active2\data
Passive_Member_1	Mongod	[hostname]:27023	C:\db1\passive1\data

The hostname used in the above table can be found out using the following command:

```
C:\>hostname
ANOC9
C:\>
```

In the following examples, the [*hostname*] need to be substituted with the value that the *hostname* command returns on your system. In our case, the value returned is ANOC9, which is used in the following examples.

Use the default (MMAPv1) storage engine in the following implementation.

Setting Up a Replica Set

In order to get the replica set up and running, you need to make all the active members up and running.

The first step is to start the first active member. Open a terminal window and create the data directory:

```
C:\>mkdir C:\db1\active1\data
C:\>
```

Connect to the mongod:

```
c:\practicalmongodb\bin>mongod --dbpath C:\db1\active1\data --port 27021 --replSet
testset/ANOC9:27021 -rest

2015-07-13T23:48:40.543-0700 I CONTROL  ** WARNING: --rest is specified without --httpinterface,
2015-07-13T23:48:40.543-0700 I CONTROL  **  enabling http interface
2015-07-13T23:48:40.543-0700 I CONTROL  Hotfix KB2731284 or later update is installed, no
                                        need to zero-out data files
2015-07-13T23:48:40.563-0700 I JOURNAL  [initandlisten] journal dir=C:\db1\active1\data\journal
2015-07-13T23:48:40.564-0700 I JOURNAL  [initandlisten] recover : no journal files present,
                                                        no recovery needed
.....................................   port=27021 dbpath=C:\db1\active1\data 64-bit
                                        host=ANOC9
2015-07-13T23:48:40.614-0700 I CONTROL  [initandlisten] targetMinOS: Windows 7/Windows
                                                        Server 2008 R2
2015-07-13T23:48:40.615-0700 I CONTROL  [initandlisten] db version v3.0.4
```

As you can see, the –replSet option specifies the name of the replica set the instance is joining and the name of one more member of the set, which in the above example is Active_Member_2.

Although you have only specified one member in the above example, multiple members can be provided by specifying comma-separated addresses like so:

```
mongod –dbpath C:\db1\active1\data –port 27021 –replset
testset/[hostname]:27022,[hostname]:27023 --rest
```

In the next step, you get the second active member up and running. Create the data directory for the second active member in a new terminal window.

```
C:\>mkdir C:\db1\active2\data
C:\>
```

Connect to mongod:

```
c:\ practicalmongodb \bin>mongod --dbpath C:\db1\active2\data --port 27022 –replSet
testset/ANOC9:27021 –rest
2015-07-13T00:39:11.599-0700 I CONTROL  ** WARNING: --rest is specified without --httpinterface,
2015-07-13T00:39:11.599-0700 I CONTROL  **          enabling http interface
2015-07-13T00:39:11.604-0700 I CONTROL  Hotfix KB2731284 or later update is installed, no
                                        need to zero-out data files
2015-07-13T00:39:11.615-0700 I JOURNAL  [initandlisten] journal dir=C:\db1\active2\data\journal
2015-07-13T00:39:11.615-0700 I JOURNAL  [initandlisten] recover : no journal files present,
                                                        no recovery needed
2015-07-13T00:39:11.664-0700 I JOURNAL  [durability] Durability thread started
2015-07-13T00:39:11.664-0700 I JOURNAL  [journal writer] Journal writer thread started
                                        rs.initiate() in the shell -- if that is not already done
```

Finally, you need to start the passive member. Open a separate window and create the data directory for the passive member.

```
C:\>mkdir C:\db1\passive1\data
C:\>
```

Connect to mongod:

```
c:\ practicalmongodb \bin>mongod --dbpath C:\db1\passive1\data --port 27023 --replSet
testset/ ANOC9:27021 –rest
2015-07-13T05:11:43.746-0700 I CONTROL  Hotfix KB2731284 or later update is installed, no
                                        need to zero-out data files
2015-07-13T05:11:43.757-0700 I JOURNAL  [initandlisten] journal dir=C:\db1\passive1\data\journal
2015-07-13T05:11:43.808-0700 I CONTROL  [initandlisten] MongoDB starting : pid=620 port=27019
                                        dbpath=C:\db1\passive1\data 64-bit host= ANOC9
.....................................................................................
2015-07-13T05:11:43.812-0700 I CONTROL  [initandlisten] options: { net: { http:
{ RESTInterfaceEnabled: true, enabled: true }, port: 27019 }, replication: { re
lSet: "testset/ ANOC9:27017" }, storage: { dbPath: "C:\db1\passive1\data" }
```

In the preceding examples, the `--rest` option is used to activate a REST interface on port +1000. Activating REST enables you to inspect the replica set status using web interface.

By the end of the above steps, you have three servers that are up and running and are communicating with each other; however the replica set is still not initialized. In the next step, you initialize the replica set and instruct each member about their responsibilities and roles.

In order to initialize the replica set, you connect to one of the servers. In this example, it is the first server, which is running on port 27021.

Open a new command prompt and connect to the mongo interface for the first server:

```
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongo ANOC9 --port 27021
MongoDB shell version: 3.0.4
connecting to: ANOC9:27021/test
>
```

Next, switch to the `admin` database.

```
> use admin
switched to db admin
>
```

Next, a configuration data structure is set up, which mentions server wise roles:

```
>cfg = {
... _id: 'testset',
... members: [
... {_id:0, host: 'ANOC9:27021'},
... {_id:1, host: 'ANOC9:27022'},
... {_id:2, host: 'ANOC9:27023', priority:0}
... ]
... }
{        "_id" : "testset",
        "members" : [
                {
                        "_id" : 0,
                        "host" : "ANOC9:27021"
                },
..........
                {
                        "_id" : 2,
                        "host" : "ANOC9:27023",
                        "priority" : 0
                }  ]}>
```

With this step the replicas set structure is configured.

You have used 0 priority when defining the role for the passive member. This means that the member cannot be promoted to primary.

The next command initiates the replica set:

```
> rs.initiate(cfg)
{ "ok" : 1}
```

Let's now view the replica set status in order to vet that it's set up correctly:

```
testset:PRIMARY> rs.status()
{
     "set" : "testset",
"date" : ISODate("2015-07-13T04:32:46.222Z")
     "myState" : 1,
"members" : [
        {
                "_id" : 0,
.........................
testset:PRIMARY>
```

The output indicates that all is OK. The replica set is now successfully configured and initialized.

Let's see how you can determine the primary node. In order to do so, connect to any of the members and issue the following and verify the primary:

```
testset:PRIMARY> db.isMaster()
{
        "setName" : "testset",
        "setVersion" : 1,
        "ismaster" : true,
        "primary" : " ANOC9:27021",
        "me" : "ANOC9:27021",
        ......................................
       "localTime" : ISODate("2015-07-13T04:36:52.365Z"),
        ..................................................
        "ok" : 1
}testset:PRIMARY>
```

Removing a Server

In this example, you will remove the secondary active member from the set. Let's connect to the secondary member mongo instance. Open a new command prompt, like so:

```
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongo ANOC9 --port 27022
MongoDB shell version: 3.0.4
connecting to: 127.0.0.1:27022/ANOC9
testset:SECONDARY>
```

Issue the following command to shut down the instance:

```
testset:SECONDARY> use admin
switched to db admin
testset:SECONDARY> db.shutdownServer()
2015-07-13T21:48:59.009-0700 I NETWORK  DBClientCursor::init call() failed server should be down...
```

Next, you need to connect to the primary member mongo console and execute the following to remove the member:

```
testset:PRIMARY> use admin
switched to db admin
testset:PRIMARY> rs.remove("ANOC9:27022")
{ "ok" : 1 }
testset:PRIMARY>
```

In order to vet whether the member is removed or not you can issue the `rs.status()` command.

Adding a Server

You will next add a new active member to the replica set. As with other members, you begin by opening a new command prompt and creating the data directory first:

```
C:\>mkdir C:\db1\active3\data
C:\>
```

Next, you start the mongod using the following command:

```
c:\practicalmongodb\bin>mongod --dbpath C:\db1\active3\data --port 27024 --replSet testset/
ANOC9:27021 --rest
..........
```

You have the new mongod running, so now you need to add this to the replica set. For this you connect to the primary's mongo console:

```
C:\>c:\practicalmongodb\bin\mongo.exe --port 27021
MongoDB shell version: 3.0.4
connecting to: 127.0.0.1:27021/test
testset:PRIMARY>
```

Next, you switch to admin db:

```
testset:PRIMARY> use admin
switched to db admin
testset:PRIMARY>
```

Finally, the following command needs to be issued to add the new mongod to the replica set:

```
testset:PRIMARY> rs.add("ANOC9:27024")
{ "ok" : 1 }
```

The replica set status can be checked to vet whether the new active member is added or not using `rs.status()`.

Adding an Arbiter to a Replica Set

In this example, you will add an arbiter member to the set. As with the other members, you begin by creating the data directory for the MongoDB instance:

```
C:\>mkdir c:\db1\arbiter\data
C:\>
```

You next start the mongod using the following command:

```
c:\practicalmongodb\bin>mongod --dbpath c:\db1\arbiter\data --port 30000 --replSet testset/
ANOC9:27021 --rest
2015-07-13T22:05:10.205-0700 I CONTROL  [initandlisten] MongoDB starting : pid=3700
port=30000 dbpath=c:\db1\arbiter\data 64-bit host=ANOC9
.....................................................
```

Connect to the primary's mongo console, switch to the admin db, and add the newly created mongod as an arbiter to the replica set:

```
C:\>c:\practicalmongodb\bin\mongo.exe --port 27021
MongoDB shell version: 3.0.4
connecting to: 127.0.0.1:27021/test
testset:PRIMARY> use admin
switched to db admin

testset:PRIMARY> rs.addArb("ANOC9:30000")
{ "ok" : 1 }
testset:PRIMARY>
```

Whether the step is successful or not can be verified using `rs.status().`

Inspecting the Status Using rs.status()

We have been referring to `rs.status()` throughout the examples above to check the replica set status. In this section, you will learn what this command is all about.

It enables you to check the status of the member whose console they are connected to and also enables them to view its role within the replica set.

The following command is issued from the primary's mongo console:

```
testset:PRIMARY> rs.status()
{
     "set" : "testset",
"date" : ISODate("2015-07-13T22:15:46.222Z")
     "myState" : 1,
"members" : [
        {
                "_id" : 0,
.........................
        "ok" : 1
testset:PRIMARY>
```

The **myState** field's value indicates the status of the member and it can have the values shown in Table 7-3.

Table 7-3. *Replica Set Status*

myState	Description
0	Phase 1, starting up
1	Primary member
2	Secondary member
3	Recovering state
4	Fatal error state
5	Phase 2, Starting up
6	Unknown state
7	Arbiter member
8	Down or unreachable
9	This state is reached when a write operation is rolled back by the secondary after transitioning from primary.
10	Members enter this state when removed from the replica set.

Hence the above command returns myState value as 1, which indicates that this is the primary member.

Forcing a New Election

The current primary server can be forced to step down using the `rs.stepDown ()` command. This force starts the election for a new primary.

This command is useful in the following scenarios:

1. When you are simulating the impact of a primary failure, forcing the cluster to fail over. This lets you test how your application responds in such a scenario.
2. When the primary server needs to be offline. This is done for either a maintenance activity or for upgrading or to investigating the server.
3. When a diagnostic process need to be run against the data structures.

The following is the output of the command when run against the testset replica set:

```
testset:PRIMARY> rs.stepDown()
2015-07-13T22:52:32.000-0700 I NETWORK  DBClientCursor::init call() failed
2015-07-13T22:52:32.005-0700 E QUERY    Error: error doing query: failed
2015-07-13T22:52:32.009-0700 I NETWORK  trying reconnect to 127.0.0.1:27021 (127.0.0.1) failed
2015-07-13T22:52:32.011-0700 I NETWORK  reconnect 127.0.0.1:27021 (127.0.0.1) ok
                                        testset:SECONDARY>
```

After execution of the command the prompt changed from testset:PRIMARY to testset:SECONDARY.

`rs.status()` can be used to check whether the `stepDown ()` is successful or not.

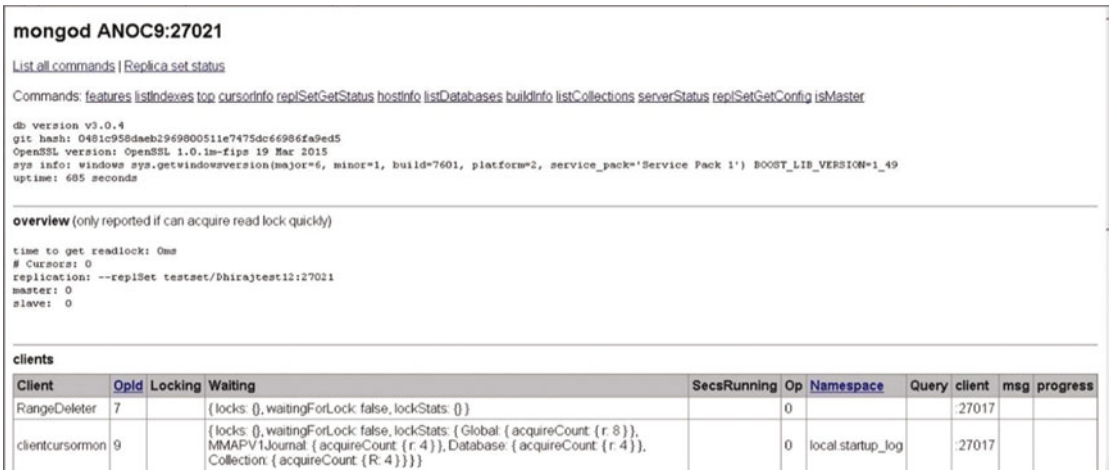
Please note the myState value it returns is 2 now, which means the "Member is operating as secondary."

Inspecting Status of the Replica Set Using a Web Interface

A web-based console is maintained by MongoDB for viewing the system status. In your example, the console can be accessed via `http://localhost:28021`.

By default the web interface port number is set to X+1000 where X is the mongod instance port number. In this chapter's example, since the primary instance is on 27021, the web interface is on port 28021.

Figure 7-13 shows a link to the replica set status. Clicking the link takes you to the replica set dashboard shown in Figure 7-14.



mongod ANOC9:27021

List all commands | Replica set status

Commands: features listIndexes top cursorInfo replSetGetStatus hostInfo listDatabases buildInfo listCollections serverStatus replSetGetConfig isMaster

```
db version v3.0.4
git hash: 0481c958daeb2969800511e7475dc66986fa9ed5
OpenSSL version: OpenSSL 1.0.1m-fips 19 Mar 2015
sys info: windows sys.getwindowsversion(major=6, minor=1, build=7601, platform=2, service_pack='Service Pack 1') BOOST_LIB_VERSION=1_49
uptime: 685 seconds
```

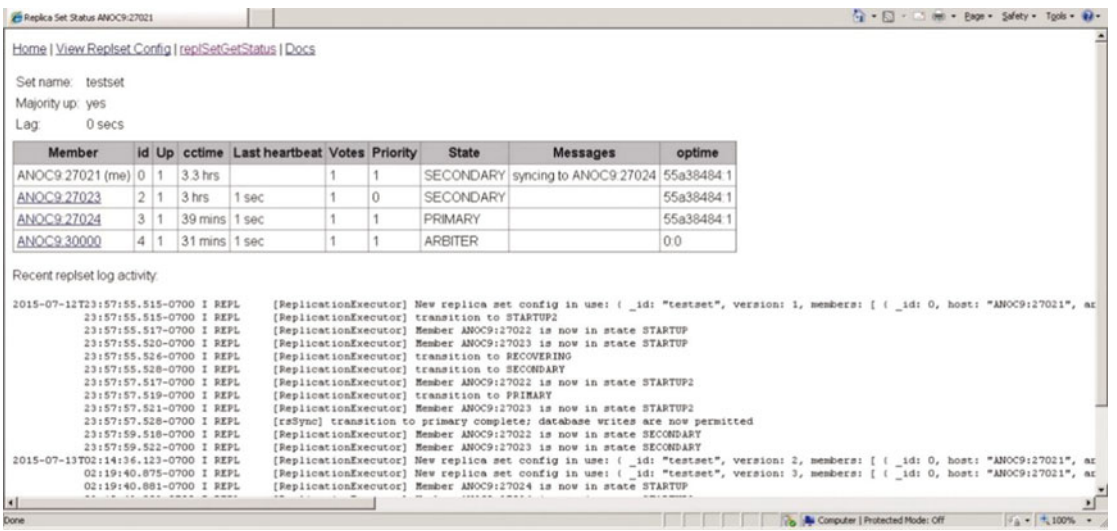
overview (only reported if can acquire read lock quickly)

```
time to get readlock: 0ms
# Cursors: 0
replication: --replSet testset/Dhirajtest12:27021
master: 0
slave:  0
```

clients

Client	OpId	Locking	Waiting	SecsRunning	Op	Namespace	Query	client	msg	progress
RangeDeleter	7		{ locks: {}, waitingForLock: false, lockStats: {} }		0			:27017		
clientcursormon	9		{ locks: {}, waitingForLock: false, lockStats: { Global: { acquireCount: { r: 8 } }, MMAPV1Journal: { acquireCount: { r: 4 } }, Database: { acquireCount: { r: 4 } }, Collection: { acquireCount: { R: 4 } } } }		0	local.startup_log		:27017		

Figure 7-13. *Web interface*



Replica Set Status ANOC9:27021

Home | View Replset Config | replSetGetStatus | Docs

Set name: testset
Majority up: yes
Lag: 0 secs

Member	id	Up	cctime	Last heartbeat	Votes	Priority	State	Messages	optime
ANOC9:27021 (me)	0	1	3.3 hrs		1	1	SECONDARY	syncing to ANOC9:27024	55a38484:1
ANOC9:27023	2	1	3 hrs	1 sec	1	0	SECONDARY		55a38484:1
ANOC9:27024	3	1	39 mins	1 sec	1	1	PRIMARY		55a38484:1
ANOC9:30000	4	1	31 mins	1 sec	1	1	ARBITER		0:0

Recent replset log activity:

```
2015-07-12T23:57:55.515-0700 I REPL     [ReplicationExecutor] New replica set config in use: { _id: "testset", version: 1, members: [ { _id: 0, host: "ANOC9:27021", ar
          23:57:55.515-0700 I REPL     [ReplicationExecutor] transition to STARTUP2
          23:57:55.517-0700 I REPL     [ReplicationExecutor] Member ANOC9:27022 is now in state STARTUP
          23:57:55.520-0700 I REPL     [ReplicationExecutor] Member ANOC9:27023 is now in state STARTUP
          23:57:55.526-0700 I REPL     [ReplicationExecutor] transition to RECOVERING
          23:57:55.528-0700 I REPL     [ReplicationExecutor] transition to SECONDARY
          23:57:57.517-0700 I REPL     [ReplicationExecutor] Member ANOC9:27022 is now in state STARTUP2
          23:57:57.519-0700 I REPL     [ReplicationExecutor] transition to PRIMARY
          23:57:57.521-0700 I REPL     [ReplicationExecutor] Member ANOC9:27023 is now in state STARTUP2
          23:57:57.528-0700 I REPL     [rsSync] transition to primary complete; database writes are now permitted
          23:57:59.518-0700 I REPL     [ReplicationExecutor] Member ANOC9:27022 is now in state SECONDARY
          23:57:59.522-0700 I REPL     [ReplicationExecutor] Member ANOC9:27023 is now in state SECONDARY
2015-07-13T02:14:36.123-0700 I REPL     [ReplicationExecutor] New replica set config in use: { _id: "testset", version: 2, members: [ { _id: 0, host: "ANOC9:27021", ar
          02:19:40.875-0700 I REPL     [ReplicationExecutor] New replica set config in use: { _id: "testset", version: 3, members: [ { _id: 0, host: "ANOC9:27021", ar
          02:19:40.881-0700 I REPL     [ReplicationExecutor] Member ANOC9:27024 is now in state STARTUP
```

Done

Figure 7-14. *Replica set status report*

Sharding

You saw in the previous section how replica sets in MongoDB are used to duplicate the data in order to protect against any adversity and to distribute the read load in order to increase the read efficiency.

MongoDB uses memory extensively for low latency database operations. When you compare the speed of reading data from memory to reading data from disk, reading from memory is approximately 100,000 times faster than reading from the disk.

In MongoDB, ideally the working set should fit in memory. The working set consists of the most frequently accessed data and indexes.

A page fault happens when data which is not there in memory is accessed by MongoDB. If there's free memory available, the OS will directly load the requested page into memory; however, in the absence of free memory, the page in memory is written to the disk and then the requested page is loaded in the memory, slowing down the process. Few operations accidentally purge large portion of the working set from the memory, leading to an adverse effect on the performance. One example is a query scanning through all documents of a database where the size exceeds the server memory. This leads to loading of the documents in memory and moving the working set out to disk.

Ensuring you have defined the appropriate index coverage for your queries during the schema design phase of the project will minimize the risk of this happening. The MongoDB explain operation can be used to provide information on your query plan and the indexes used.

MongoDB's `serverStatus` command returns a workingSet document that provides an estimate of the instance's working set size. The Operations team can track how many pages the instance accessed over a given period of time and the elapsed time between the working set's oldest and newest document. Tracking all these metrics, it's possible to detect when the working set will be hitting the current memory limit, so proactive actions can be taken to ensure the system is scaled well enough to handle that.

In MongoDB, the scaling is handled by scaling out the data horizontally (i.e. partitioning the data across multiple commodity servers), which is also called sharding (horizontal scaling).

Sharding addresses the challenges of scaling to support large data sets and high throughput by horizontally dividing the datasets across servers where each server is responsible for handling its part of data and no one server is burdened. These servers are also called shards.

Every shard is an independent database. All the shards collectively make up a single logical database.

Sharding reduces the operations count handled by each shard. For example, when data is inserted, only the shards responsible for storing those records need to be accessed.

The processes that need to be handled by each shard reduce as the cluster grows because the subset of data that the shard holds reduces. This leads to an increase in the throughput and capacity horizontally.

Let's assume you have a database that is 1TB in size. If the number of shards is 4, you will have approximately 265GB of data handled by each shard, whereas if the number of shards is increased to 40, only 25GB of data will be held on each shard.

Figure 7-15 depicts how a collection that is sharded will appear when distributed across three shards.

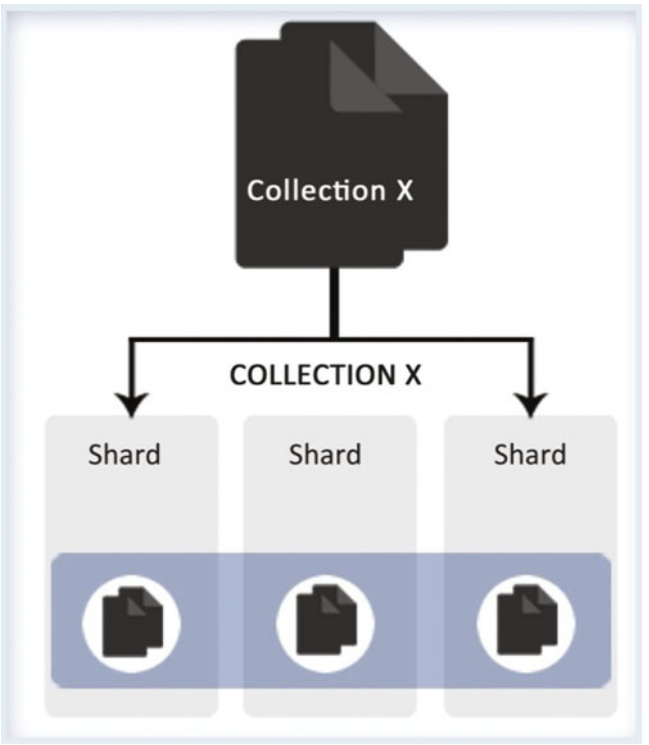


Figure 7-15. *Sharded collection across three shards*

Although sharding is a compelling and powerful feature, it has significant infrastructure requirements and it increases the complexity of the overall deployment. So you need to understand the scenarios where you might consider using sharding.

Use sharding in the following instances:

- The size of the dataset is huge and it has started challenging the capacity of a single system.
- Since memory is used by MongoDB for quickly fetching data, it becomes important to scale out when the active work set limits are set to reach.
- If the application is write-intensive, sharding can be used to spread the writes across multiple servers.

Sharding Components

You will next look at the components that enable sharding in MongoDB. Sharding is enabled in MongoDB via sharded clusters.

The following are the components of a sharded cluster:

- Shards
- mongos
- Config servers

The shard is the component where the actual data is stored. For the sharded cluster, it holds a subset of data and can either be a mongod or a replica set. All shard's data combined together forms the complete dataset for the sharded cluster.

Sharding is enabled per collection basis, so there might be collections that are not sharded. In every sharded cluster there's a primary shard where all the unsharded collections are placed in addition to the sharded collection data.

When deploying a sharded cluster, by default the first shard becomes the primary shard although it's configurable. See Figure 7-16.

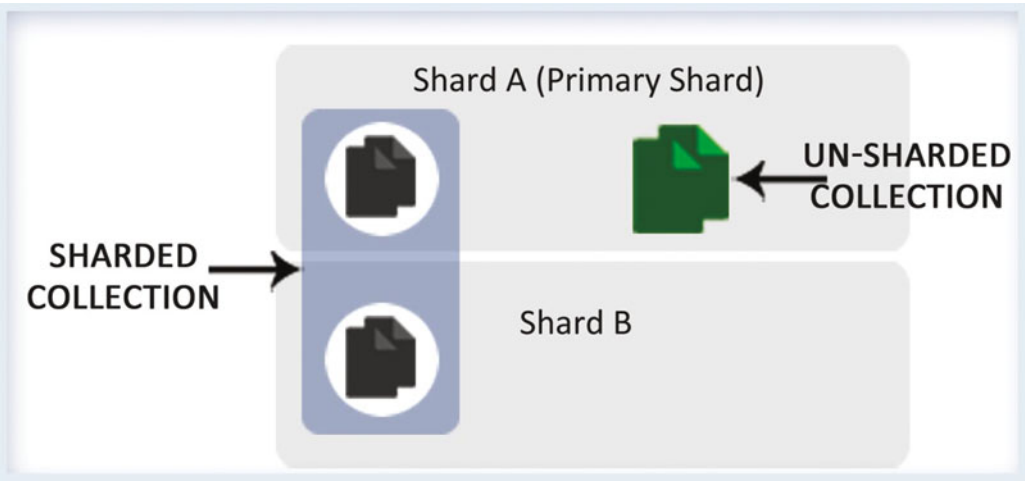


Figure 7-16. *Primary shard*

Config servers are special mongods that hold the sharded cluster's metadata. This metadata depicts the sharded system state and organization.

The config server stores data for a single sharded cluster. The config servers should be available for the proper functioning of the cluster.

One config server can lead to a cluster's single point of failure. For production deployment it's recommended to have at least three config servers, so that the cluster keeps functioning even if one config server is not accessible.

A config server stores the data in the config database, which enables routing of the client requests to the respective data. This database should not be updated.

MongoDB writes data to the config server only when the data distribution has changed for balancing the cluster.

The mongos act as the routers. They are responsible for routing the read and write request from the application to the shards.

An application interacting with a mongo database need not worry about how the data is stored internally on the shards. For them, it's transparent because it's only the mongos they interact with. The mongos, in turn, route the reads and writes to the shards.

The mongos cache the metadata from config server so that for every read and write request they don't overburden the config server.

However, in the following cases, the data is read from the config server:

- Either an existing mongos has restarted or a new mongos has started for the first time.
- Migration of chunks. We will explain chunk migration in detail later.

Data Distribution Process

You will next look at how the data is distributed among the shards for the collections where sharding is enabled. In MongoDB, the data is sharded or distributed at the collection level. The collection is partitioned by the shard key.

Shard Key

Any indexed single/compound field that exists within all documents of the collection can be a shard key. You specify that this is the field basis which the documents of the collection need to be distributed. Internally, MongoDB divides the documents based on the value of the field into chunks and distributes them across the shards.

There are two ways MongoDB enables distribution of the data: range-based partitioning and hash-based partitioning.

Range-Based Partitioning

In range-based partitioning, the shard key values are divided into ranges. Say you consider a `timestamp` field as the shard key. In this way of partitioning, the values are considered as a straight line starting from a Min value to Max value where Min is the starting period (say, 01/01/1970) and Max is the end period (say, 12/31/9999). Every document in the collection will have timestamp value within this range only, and it will represent some point on the line.

Based on the number of shards available, the line will be divided into ranges, and documents will be distributed based on them.

In this scheme of partitioning, shown in Figure 7-17, the documents where the values of the shard key are nearby are likely to fall on the same shard. This can significantly improve the performance of the range queries.

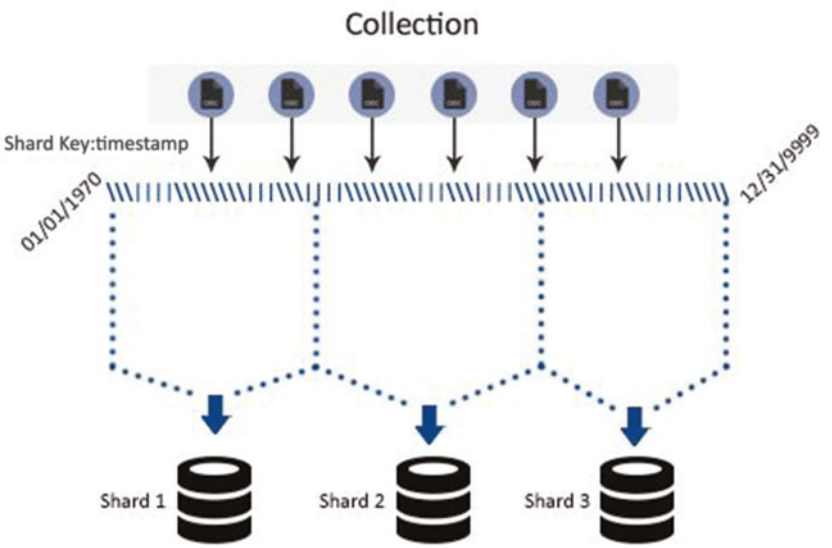


Figure 7-17. *Range-based partitioning*

However, the disadvantage is that it can lead to uneven distribution of data, overloading one of the shards, which may end up receiving majority of the requests, whereas the other shards remain underloaded, so the system will not scale properly.

Hash-Based Partitioning

In hash-based partitioning, the data is distributed on the basis of the hash value of the shard field. If selected, this will lead to a more random distribution compared to range-based partitioning.

It's unlikely that the documents with close shard key will be part of the same chunk. For example, for ranges based on the hash of the `_id` field, there will be a straight line of hash values, which will again be partitioned on basis of the number of shards. On the basis of the hash values, the documents will lie in either of the shards. See Figure 7-18.

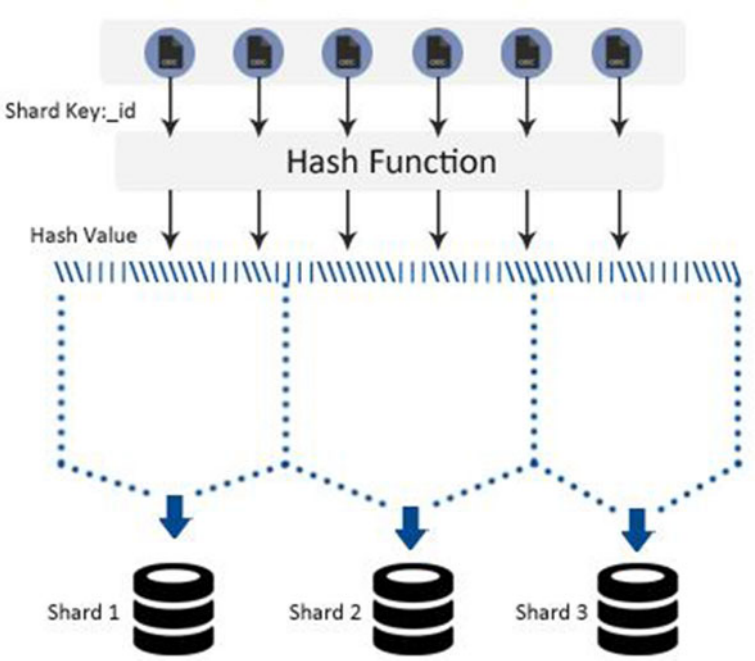


Figure 7-18. *Hash-based partitioning*

In contrast to range-based partitioning, this ensures that the data is evenly distributed, but it happens at the cost of efficient range queries.

Chunks

The data is moved between the shards in form of chunks. The shard key range is further partitioned into sub-ranges, which are also termed as chunks. See Figure 7-19.

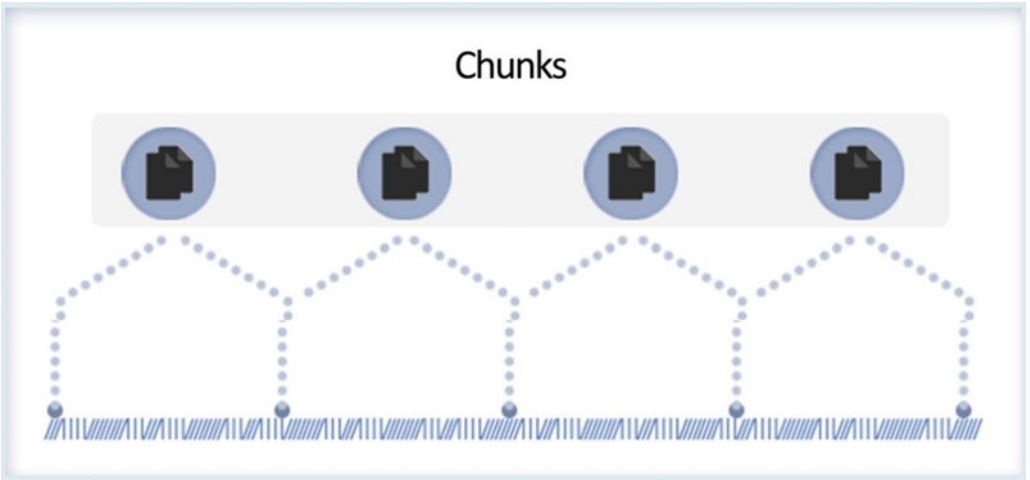


Figure 7-19. *Chunks*

For a sharded cluster, 64MB is the default chunk size. In most situations, this is an apt size for chunk slitting and migration.

Let's discuss the execution of sharding and chunks with an example. Say you have a blog posts collection which is sharded on the field `date`. This implies that the collection will be split up on the basis of the `date` field values. Let's assume further that you have three shards. In this scenario the data might be distributed across shards as follows:

Shard #1: Beginning of time up to July 2009

Shard #2: August 2009 to December 2009

Shard #3: January 2010 to through the end of time

In order to retrieve documents from January 1, 2010 until today, the query is sent to mongos.
In this scenario,

1. The client queries mongos.
2. The mongos know which shards have the data, so mongos sends the queries to Shard #3.
3. Shard #3 executes the query and returns the results to mongos.
4. Mongos combines the data received from various shards, which in this case is Shard #3 only, and returns the final result back to the client.

The application doesn't need to be sharding-aware. It can query the mongos as though it's a normal mongod.

Let's consider another scenario where you insert a new document. The new document has today's date. The sequences of events are as follows:

1. The document is sent to the mongos.
2. Mongos checks the date and on basis of that, sends the document to Shard #3.
3. Shard #3 inserts the document.

From a client's point of view, this is again identical to a single server setup.

Role of ConfigServers in the Above Scenario

Consider a scenario where you start getting insert requests for millions of documents with the date of September 2009. In this case, Shard #2 begins to get overloaded.

The config server steps in once it realizes that Shard #2 is becoming too big. It will split the data on the shard and start migrating it to other shards. After the migration is completed, it sends the updated status to the mongos. So now Shard #2 has data from August 2009 until September 18, 2009 and Shard #3 contains data from September 19, 2009 until the end of time.

When a new shard is added to the cluster, it's the config server's responsibility to figure out what to do with it. The data may need to be immediately migrated to the new shard, or the new shard may need to be in reserve for some time. In summary, the config servers are the brains. Whenever any data is moved around, the config servers let the mongos know about the final configuration so that the mongos can continue doing proper routing.

Data Balancing Process

You will next look at how the cluster is kept balanced (i.e. how MongoDB ensures that all the shards are equally loaded).

The addition of new data or modification of existing data, or the addition or removal of servers, can lead to imbalance in the data distribution, which means either one shard is overloaded with more chunks and the other shards have less number of chunks, or it can lead to an increase in the chunk size, which is significantly greater than the other chunks.

MongoDB ensures balance with the following background processes:

- Chunk splitting
- Balancer

Chunk Splitting

Chunk splitting is one of the processes that ensures the chunks are of the specified size. As you have seen, a shard key is chosen and it is used to identify how the documents will be distributed across the shards. The documents are further grouped into chunks of 64MB (default and is configurable) and are stored in the shards based on the range it is hosting.

If the size of the chunk changes due to an insert or update operation, and exceeds the default chunk size, then the chunk is split into two smaller chunks by the mongos. See Figure 7-20.

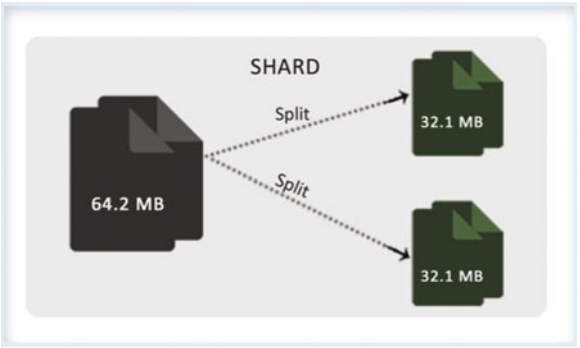


Figure 7-20. *Chunk splitting*

This process keeps the chunks within a shard of the specified size or lesser than that (i.e. it ensures that the chunks are of the configured size).

Insert and update operations trigger splits. The split operation leads to modification of the data in the config server as the metadata is modified. Although splits don't lead to migration of data, this operation can lead to an unbalance of the cluster with one shard having more chunks compared to another.

Balancer

Balancer is the background process that is used to ensure that all of the shards are equally loaded or are in a balanced state. This process manages chunk migrations.

Splitting of the chunk can cause imbalance. The addition or removal of documents can also lead to a cluster imbalance. In a cluster imbalance, balancer is used, which is the process of distributing data evenly.

When you have a shard with more chunks as compared to other shards, then the chunks balancing is done automatically by MongoDB across the shards. This process is transparent to the application and to you.

Any of the mongos within the cluster can initiate the balancer process. They do so by acquiring a lock on the config database of the config server, as balancer involves migration of chunks from one shard to another, which can lead to a change in the metadata, which will lead to change in the config server database. The balancer process can have huge impact on the database performance, so it can either

1. Be configured to start the migration only when the migration threshold has reached. The migration threshold is the difference in the number of maximum and minimum chunks on the shards. Threshold is shown in Table 7-4.

Table 7-4. *Migration Threshold*

Number of Chunks	Migration Threshold
< 20	2
21-80	4
>80	8

2. Or it can be scheduled to run in a time period that will not impact the production traffic.

The balancer migrates one chunk at a time (see Figure 7-21) and follows these steps:

1. The `moveChunk` command is sent to the source shard.
2. An internal `moveChunk` command is started on the source where it creates the copy of the documents within the chunk and queues it. In the meantime, any operations for that chunk are routed to the source by the mongos because the config database is not yet changed and the source will be responsible for serving any read/write request on that chunk.
3. The destination shard starts receiving the copy of the data from the source.
4. Once all of the documents in the chunks have been received by the destination shard, the synchronization process is initiated to ensure that all changes that have happened to the data while migration are updated at the destination shard.
5. Once the synchronization is completed, the next step is to update the metadata with the chunk's new location in the config database. This activity is done by the destination shard that connects to the config database and carries out the necessary updates.
6. Post successful completion of all the above, the document copy that is maintained at the source shard is deleted.

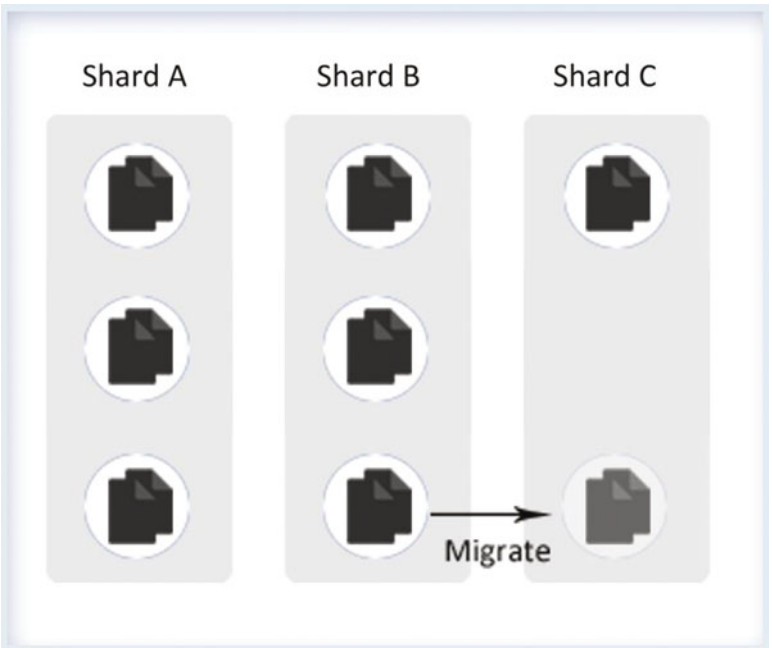


Figure 7-21. *Chunk migration*

If in the meanwhile the balancer needs additional chunk migration from the source shard, it can start with the new migration without even waiting for the deletion step to finish for the current migration.

In case of any error during the migration process, the process is aborted by the balancer, leaving the chunks on the original shard. On successful completion of the process, the chunk data is removed from the original shard by MongoDB.

Addition or removal of shards can also lead to cluster imbalance. When a new shard is added, data migration to the shard is started immediately. However, it takes time for the cluster to be balanced.

When a shard is removed, the balancer ensures that the data is migrated to the other shards and the metadata information is updated. Post completion of the two activities, the shard is removed safely.

Operations

You will next look at how the read and write operations are performed on the sharded cluster. As mentioned, the config servers maintain the cluster metadata. This data is stored in the config database. This data of the config database is used by the mongos to service the application read and write requests.

The data is cached by the mongos instances, which is then used for routing write and read operations to the shards. This way the config servers are not overburdened.

The mongos will only read from the config servers in the following scenarios:

- The mongos has started for first time or
- An existing mongos has restarted or
- After chunk migration when the mongos needs to update its cached metadata with the new cluster metadata.

Whenever any operation is issued, the first step that the mongos need to do is to identify the shards that will be serving the request. Since the shard key is used to distribute data across the sharded cluster, if the operation is using the shard key field, then based on that specific shards can be targeted.

If the shard key is `employeeid`, the following things can happen:

1. If the `find` query contains the `employeeid` field, then to satiate the query, only specific shards will be targeted by the mongos.
2. If a single update operation uses `employeeid` for updating the document, the request will be routed to the shard holding that employee data.

However, if the operation is not using the shard key, then the request is broadcast to all the shards. Generally a multi-update or remove operation is targeted across the cluster.

While querying the data, there might be scenarios where in addition to identifying the shards and getting the data from them, the mongos might need to work on the data returned from various shards before sending the final output to the client.

Say an application has issued a `find()` request with `sort()`. In this scenario, the mongos will pass the `$orderby` option to the shards. The shards will fetch the data from their data set and will send the result in an ordered manner. Once the mongos has all the shard's sorted data, it will perform an incremental merge sort on the entire data and then return the final output to the client.

Similar to sort are the aggregation functions such as `limit()`, `skip()`, etc., which require mongos to perform operations post receiving the data from the shards and before returning the final result set to the client.

The mongos consumes minimal system resources and has no persistent state. So if the application requirement is a simple `find ()` queries that can be solely met by the shards and needs no manipulation at the mongos level, you can run the mongos on the same system where your application servers are running.

Implementing Sharding

In this section, you will learn to configure sharding in one machine on a Windows platform.

You will keep the example simple by using only two shards. In this configuration, you will be using the services listed in Table 7-5.

Table 7-5. *Sharding Cluster Configuration*

Component	Type	Port	Datafile path
Shard Controller	Mongos	27021	-
Config Server	Mongod	27022	C:\db1\config\data
Shard0	Mongod	27023	C:\db1\shard1\data
Shard1	Mongod	27024	C:\db1\shard2\data

You will be focusing on the following:

1. Setting up a sharded cluster.
2. Creating a database and collection, and enable sharding on the collection.
3. Using the import command to load data in the sharded collection.
4. Distributed data amongst the shards.
5. Adding and removing shards from the cluster and checking how data is distributed automatically.

Setting the Shard Cluster

In order to set up the cluster, the first step is to set up the configuration server. Enter the following code in a new terminal window to create the `data` directory for the config server and start the mongod:

```
C:\> mkdir C:\db1\config\data
C:\>CD C:\practicalmongodb\bin
C:\ practicalmongodb\bin>mongod --port 27022 --dbpath C:\db1\config\data --configsvr

2015-07-13T23:02:41.982-0700 I JOURNAL  [journal writer] Journal writer thread started
2015-07-13T23:02:41.984-0700 I CONTROL  [initandlisten] MongoDB starting : pid=3084
port=27022 dbpath=C:\db1\config\data master=1 64-bit host=ANOC9
.....................................
2015-07-13T23:02:42.066-0700 I REPL     [initandlisten] ******
2015-07-13T03:02:42.067-0700 I NETWORK  [initandlisten] waiting for connections on port 27022
```

Next, start the mongos. Type the following in a new terminal window:

```
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongos --configdb localhost:27022 --port 27021 --chunkSize 1
2015-07-13T23:06:07.246-0700 W SHARDING running with 1 config server should be done only for
testing purposes and is not recommended for production
.....................................................................
2015-07-13T23:09:07.464-0700 I SHARDING [Balancer] distributed lock 'balancer/
ANOC9:27021:1429783567:41' unlocked
```

You now have the shard controller (i.e. the mongos) up and running.

If you switch to the window where the config server has been started, you will find a registration of the shard server to the config server.

In this example you have used chunk size of 1MB. Note that this is not ideal in a real-life scenario since the size is less than 4MB (a document's maximum size). However, this is just for demonstration purpose since this creates the necessary amount of chunks without loading a large amount of data. The chunkSize is 128MB by default unless otherwise specified.

Next, bring up the shard servers, Shard0 and Shard1.

Open a fresh terminal window. Create the `data` directory for the first shard and start the mongod:

```
C:\>mkdir C:\db1\shard0\data
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongod --port 27023 --dbpath c:\db1\shard0\data –shardsvr
2015-07-13T23:14:58.076-0700 I CONTROL  [initandlisten] MongoDB starting : pid=1996
port=27023 dbpath=c:\db1\shard0\data 64-bit host=ANOC9
.....................................................................
2015-07-13T23:14:58.158-0700 I NETWORK  [initandlisten] waiting for connections on port 27023
```

Open fresh terminal window. Create the `data` directory for the second shard and start the mongod:

```
C:\>mkdir c:\db1\shard1\data
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongod --port 27024 --dbpath C:\db1\shard1\data --shardsvr
2015-07-13T23:17:01.704-0700 I CONTROL  [initandlisten] MongoDB starting : pid=3672
port=27024 dbpath=C:\db1\shard1\data 64-bit host=ANOC9
2015-07-13T23:17:01.704-0700 I NETWORK  [initandlisten] waiting for connections on port 27024
```

All the servers relevant for the setup are up and running by the end of the above step. The next step is to add the shards information to the shard controller.

The mongos appears as a complete MongoDB instance to the application in spite of actually not being a full instance. The mongo shell can be used to connect to the mongos to perform any operation on it.

Open the mongos mongo console:

```
C:\>cd c:\practicalmongodb\bin
c:\ practicalmongodb\bin>mongo localhost:27021
MongoDB shell version: 3.0.4
connecting to: localhost:27021/test
mongos>
```

Switch to the admin database:

```
mongos> use admin
switched to db admin
mongos>
```

Add the shards information by running the following commands:

```
mongos> db.runCommand({addshard:"localhost:27023",allowLocal:true})
{ "shardAdded" : "shard0000", "ok" : 1 }
mongos> db.runCommand({addshard:"localhost:27024",allowLocal:true})
{ "shardAdded" : "shard0001", "ok" : 1 }
mongos>
```

This activates the two shard servers.

The next command checks the shards:

```
mongos> db.runCommand({listshards:1})
{
        "shards" : [
                {
                        "_id" : "shard0000",
                        "host" : "localhost:27023"
                },                {
                        "_id" : "shard0001",
                        "host" : "localhost:27024"
                }
        ],        "ok" : 1}
```

Creating a Database and Shard Collection

In order to continue further with the example, you will create a database named testdb and a collection named testcollection, which you will be sharding on the key testkey.

Connect to the mongos console and issue the following command to get the database:

```
mongos> testdb=db.getSisterDB("testdb")
testdb
```

Next, enabling sharding at database level for testdb:

```
mongos> db.runCommand({enableSharding system: "testdb"})
{ "ok" : 1 }
mongos>
```

Next, specify the collection that needs to be sharded and the key on which the collection will be sharded:

```
mongos> db.runCommand({shardcollection: "testdb.testcollection", key: {testkey:1}})
{ "collectionsharded" : "testdb.testcollection", "ok" : 1 }
mongos>
```

With the completion of the above steps you now have a sharded cluster set up with all components up and running. You have also created a database and enabled sharding on the collection.

Next, import data into the collection so that you can check the data distribution on the shards.

You will be using the import command to load data in the `testcollection`. Connect to a new terminal window and execute the following:

```
C:\>cd C:\practicalmongodb\bin
C:\practicalmongodb\bin>mongoimport --host ANOC9 --port 27021 --db testdb --collection
testcollection --type csv --file c:\mongoimport.csv --headerline
2015-07-13T23:17:39.101-0700    connected to: ANOC9:27021
2015-07-13T23:17:42.298-0700    [###############.........] testdb.testcollection 1.1 MB/1.9 MB (59.6%)
2015-07-13T23:17:44.781-0700    imported 100000 documents
```

The `mongoimport.csv` consists of two fields. The first is the `testkey`, which is a randomly generated number. The second field is a text field; it is used to ensure that the documents occupy a sufficient number of chunks, making it feasible to use the sharding mechanism.

This inserts 100,000 objects in the collection.

In order to vet whether the records are inserted or not, connect to the mongo console of the mongos and issue the following command:

```
C:\Windows\system32>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongo localhost:27021
MongoDB shell version: 3.0.4
connecting to: localhost:27021/test
mongos> use testdb
switched to db testdb
mongos> db.testcollection.count()
100000
mongos>
```

Next, connect to the consoles of the two shards (Shard0 and Shard1) and look at how the data is distributed. Open a new terminal window and connect to Shard0's console:

```
C:\>cd C:\practicalmongodb\bin
C:\ practicalmongodb\bin>mongo localhost:27023
MongoDB shell version: 3.0.4
connecting to: localhost:27023/test
```

Switch to `testdb` and issue the `count()` command to check number of documents on the shard:

```
> use testdb
switched to db testdb
> db.testcollection.count()
57998
```

Next, open a new terminal window, connect to Shard1's console, and follow the steps as above (i.e. switch to `testdb` and check the count of `testcollection` collection):

```
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongo localhost:27024
MongoDB shell version: 3.0.4
connecting to: localhost:27024/test
> use testdb
switched to db testdb
> db.testcollection.count()
42002
>
```

You might see a difference in the document's number in each shard when you run the above command for some time. When the documents are loaded, all of the chunks are placed on one shard by the mongos. In time the shard set is rebalanced by distributing the chunks evenly across all the shards.

Adding a New Shard

You have a sharded cluster set up and you also have sharded a collection and looked at how the data is distributed amongst the shards. Next, you'll add a new shard to the cluster so that the load is spread out a little more.

You will be repeating the steps mentioned above. Begin by creating a data directory for the new shard in a new terminal window:

```
c:\>mkdir c:\db1\shard2\data
```

Next, start the mongod at port 27025:

```
c:\>cd c:\practicalmongodb\bin
c:\ practicalmongodb\bin>mongod --port 27025 --dbpath C:\db1\shard2\data --shardsvr
2015-07-13T23:25:49.103-0700 I CONTROL  [initandlisten] MongoDB starting : pid=3744
port=27025 dbpath=C:\db1\shard2\data 64-bit host=ANOC9
.............................
2015-07-13T23:25:49.183-0700 I NETWORK  [initandlisten] waiting for connections on port 27025
```

Next, the new shard server will be added to the shard cluster. In order to configure it, open the mongos mongo console in a new terminal window:

```
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongo localhost:27021
MongoDB shell version: 3.0.4
connecting to: localhost:27021/test
mongos>
```

Switch to the `admin` database and run the `addshard` command. This command adds the shard server to the sharded cluster.

```
mongos> use admin
switched to db admin
mongos> db.runCommand({addshard: "localhost:27025", allowlocal: true})
{ "shardAdded" : "shard0002", "ok" : 1 }
mongos>
```

In order to vet whether the addition is successful or not, run the `listshards` command:

```
mongos> db.runCommand({listshards:1})
{
        "shards" : [
                {
                        "_id" : "shard0000",
                        "host" : "localhost:27023"
                },
```

```
                {
                        "_id" : "shard0001",
                        "host" : "localhost:27024"
                },
                {
                        "_id" : "shard0002",
                        "host" : "localhost:27025"
                }
        ],
        "ok" : 1
}
```

Next, check how the `testcollection` data is distributed. Connect to the new shard's console in a new terminal window:

```
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongo localhost:27025
MongoDB shell version: 3.0.4
connecting to: localhost:27025/test
```

Switch to `testdb` and check the collections listed on the shard:

```
> use testdb
switched to db testdb
> show collections
system.indexes
testcollection
```

Issue a `testcollection.count` command three times:

```
> db.testcollection.count()
6928
> db.testcollection.count()
12928
> db.testcollection.count()
16928
```

Interestingly, the number of items in the collection is slowly going up. The mongos is rebalancing the cluster.

With time, the chunks will be migrated from the shard servers Shard0 and Shard1 to the newly added shard server, Shard2, so that the data is evenly distributed across all the servers. Post completion of this process the config server metadata is updated. This is an automatic process and it happens even if there's no new data addition in the testcollection. This is one of the important factors you need to consider when deciding on the chunk size.

If the value of `chunkSize` is very large, you will end up having less even data distribution. The data is more evenly distributed when the `chunkSize` is smaller.

Removing a Shard

In the following example, you will see how to remove a shard server. For this example, you will be removing the server you added in the above example.

In order to initiate the process, you need to log on to the mongos console, switch to the `admin` db, and execute the following command to remove the shard from the shard cluster:

```
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongo localhost:27021
MongoDB shell version: 3.0.4
connecting to: localhost:27021/test
mongos> use admin
switched to db admin
mongos> db.runCommand({removeShard: "localhost:27025"})
{
        "msg" : "draining started successfully",
        "state" : "started",
        "shard" : "shard0002",
        "ok" : 1
}
mongos>
```

As you can see, the `removeShard` command returns a message. One of the message fields is `state`, which indicates the process state. The message also states that the draining process has started. This is indicated by the field `msg`.

You can reissue the `removeShard` command to check the progress:

```
mongos> db.runCommand({removeShard: "localhost:27025"})
{
        "msg" : "draining ongoing",
        "state" : "ongoing",
        "remaining" : {
                "chunks" : NumberLong(2),
                "dbs" : NumberLong(0)
        },
        "ok" : 1
}
mongos>
```

The response tells you the number of chunks and databases that still need to be drained from the server. If you reissue the command and the process is terminated, the output of the command will depict the same.

```
mongos> db.runCommand({removeShard: "localhost:27025"})
{
        "msg" : "removeshard completed successfully",
        "state" : "completed",
        "shard" : "shard0002",
        "ok" : 1
}
mongos>
```

You can use the `listshards` to vet whether `removeShard` was successful or not.

As you can see, the data is successfully migrated to the other shards, so you can delete the storage files and terminate the Shard2 mongod process.

This ability to modify the shard cluster without going offline is one of the critical components of MongoDB, which enables it to support highly available, highly scalable, large capacity data stores.

Listing the Sharded Cluster Status

The `printShardingStatus()` command gives lots of insight into the sharding system internals.

```
mongos> db.printShardingStatus()
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "version" : 3,
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("52fb7a8647e47c5884749a1a")
}
  shards:
        {  "_id" : "shard0000",  "host" : "localhost:27023" }
        {  "_id" : "shard0001",  "host" : "localhost:27024" }
balancer:
       Currently enabled:  yes
       Currently running:  no
       Failed balancer rounds in last 5 attempts:  0
       Migration Results for the last 24 hours:
              17 : Success
 databases:
       {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
       {  "_id" : "testdb",  "partitioned" : true,  "primary" : "shard0000" }
    ..............
```

The output lists the following:

- All of the shard servers of the shard cluster
- The configurations of each sharded database/collection
- All of the chunks of the sharded dataset

Important information that can be obtained from the above command is the sharding keys range, which is associated with each chunk. This also shows where specific chunks are stored (on which shard server). The output can be used to analyse the shard server's keys and chunks distribution.

Controlling Collection Distribution (Tag-Based Sharding)

In the previous section, you saw how data distribution happens. In this section, you will learn about tag-based sharding. This feature was introduced in version 2.2.0.

Tagging gives operators control over which collections go to which shard.

In order to understand tag-based sharding, let's set up a sharded cluster. You will be using the shard cluster created above. For this example, you need three shards, so you will add Shard2 again to the cluster.

Prerequisite

You will start the cluster first. Just to reiterate, follow these steps.

1. Start the config server. Enter the following command in a new terminal window (if it's not already running):

```
C:\> mkdir C:\db1\config\data
C:\>cd c:\practicalmongodb\bin
C:\practicalmongodb\bin>mongod --port 27022 --dbpath C:\db\config\data --configsvr
```

2. Start the mongos. Enter the following command in a new terminal window (if it's not already running):

```
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongos --configdb localhost:27022 --port 27021
```

3. You will start the shard servers next.

Start Shard0. Enter the following command in a new terminal window (if it's not already running):

```
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongod --port 27023 --dbpath c:\db1\shard0\data --shardsvr
```

Start Shard1. Enter the following command in a new terminal window (if it's not already running):

```
C:\>cd c:\practicalmongodb\bin
C:\practicalmongodb\bin>mongod --port 27024 --dbpath c:\db1\shard1\data –shardsvr
```

Start Shard2. Enter the following command in a new terminal window (if it's not already running):

```
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongod --port 27025 --dbpath c:\db1\shard2\data –shardsvr
```

Since you have removed Shard2 from the sharded cluster in the earlier example, you must add Shard2 to the sharded cluster because for this example you need three shards.

In order to do so, you need to connect to the mongos. Enter the following commands:

```
C:\Windows\system32>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongo localhost:27021
MongoDB shell version: 3.0.4
connecting to: localhost:27021/test
mongos>
```

Before the shard is added to the cluster you need to delete the `testdb` database:

```
mongos> use testdb
switched to db testdb
mongos> db.dropDatabase()
{ "dropped" : "testdb", "ok" : 1 }
mongos>
```

Next, add the Shard2 shard using the following steps:

```
mongos> use admin
switched to db admin
mongos> db.runCommand({addshard: "localhost:27025", allowlocal: true})
{ "shardAdded" : "shard0002", "ok" : 1 }
mongos>
```

If you try adding the removed shard without removing the `testdb` database, it will give the following error:

```
mongos>db.runCommand({addshard: "localhost:27025", allowlocal: true})
{
        "ok" : 0,
        "errmsg" : "can't add shard localhost:27025 because a local database 'testdb' exists
        in another shard0000:localhost:27023"}
```

In order to ensure that all the three shards are present in the cluster, run the following command:

```
mongos> db.runCommand({listshards:1})
{
        "shards" : [
                {
                        "_id" : "shard0000",
                        "host" : "localhost:27023"
                },     {
                        "_id" : "shard0001",
                        "host" : "localhost:27024"
                },    {
                        "_id" : "shard0002",
                        "host" : "localhost:27025"
                }
        ],     "ok" : 1}
```

Tagging

By the end of the above steps you have your sharded cluster with a config server, three shards, and a mongos up and running. Next, connect to the mongos at 30999 port and configdb at 27022 in a new terminal window:

```
C:\ >cd c:\practicalmongodb\bin
c:\ practicalmongodb\bin>mongos --port 30999 --configdb localhost:27022
2015-07-13T23:24:39.674-0700 W SHARDING running with 1 config server should be done only for
testing purposes and is not recommended for production
.....................................
2015-07-13T23:24:39.931-0700 I SHARDING [Balancer] distributed lock 'balancer /ANOC9:30999:
1429851279:41' unlocked..
```

Next, start a new terminal window, connect to the mongos, and enable sharding on the collections:

```
C:\ >cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongo localhost:27021
MongoDB shell version: 3.0.4
connecting to: localhost:27021/test
mongos> show dbs
admin   (empty)
config  0.016GB
testdb  0.078GB
mongos> conn=new Mongo("localhost:30999")
connection to localhost:30999
mongos> db=conn.getDB("movies")
movies
mongos> sh.enableSharding("movies")
{ "ok" : 1 }
mongos> sh.shardCollection("movies.drama", {originality:1})
{ "collectionsharded" : "movies.hindi", "ok" : 1 }
mongos> sh.shardCollection("movies.action", {distribution:1})
{ "collectionsharded" : "movies.action", "ok" : 1 }
mongos> sh.shardCollection("movies.comedy", {collections:1})
{ "collectionsharded" : "movies.comedy", "ok" : 1 }
mongos>
```

The steps are as follows:

1. Connect to the mongos console.
2. View the running databases connected to the mongos instance running at port 30999.
3. Get reference to the database `movies`.
4. Enable sharding of the database `movies`.
5. Shard the collection `movies.drama` by shard key `originality`.
6. Shard the collection `movies.action` by shard key `distribution`.
7. Shard the collection `movies.comedy` by shard key `collections`.

Next, insert some data in the collections, using the following sequence of commands:

```
mongos>for(var i=0;i<100000;i++){db.drama.insert({originality:Math.random(), count:i,
time:new Date()});}
mongos>for(var i=0;i<100000;i++){db.action.insert({distribution:Math.random(),
count:i, time:new Date()});}
mongos>for(var i=0;i<100000;i++) {db.comedy.insert({collections:Math.random(), count:i,
time:new Date()});}
mongos>
```

By the end of the above step you have three shards and three collections with sharding enabled on the collections. Next you will see how data is distributed across the shards.

Switch to configdb:

```
mongos> use config
switched to db config
mongos>
```

You can use `chunks.find` to look at how the chunks are distributed:

```
mongos> db.chunks.find({ns:"movies.drama"}, {shard:1, _id:0}).sort({shard:1})
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
mongos> db.chunks.find({ns:"movies.action"}, {shard:1, _id:0}).sort({shard:1})
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
mongos> db.chunks.find({ns:"movies.comedy"}, {shard:1, _id:0}).sort({shard:1})
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0001" }
```

```
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
mongos>
```

As you can see, the chunks are pretty evenly spread out amongst the shards. See Figure 7-22.

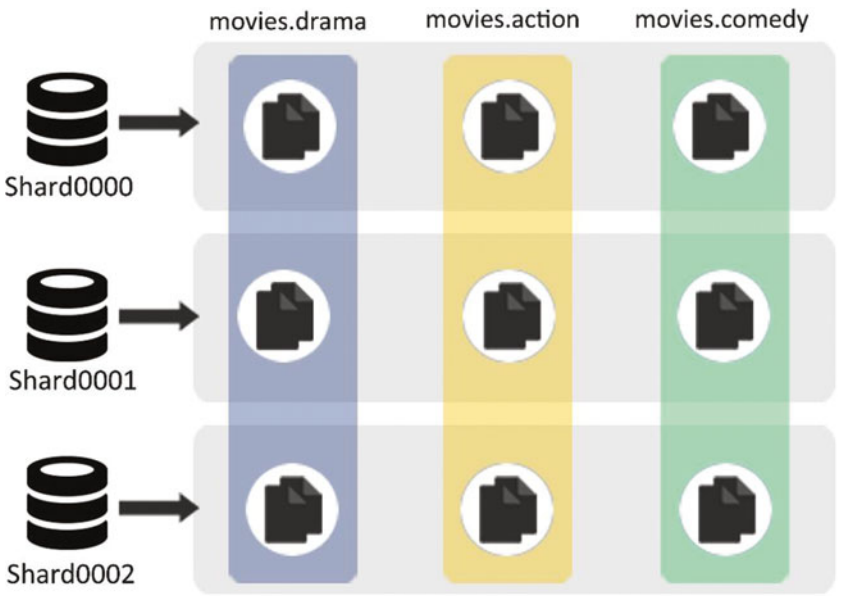


Figure 7-22. *Distribution without tagging*

Next, you will use tags to separate the collections. The intent of this is to have one collection per shard (i.e. your goal is to have the chunk distribution shown in Table 7-6).

Table 7-6. *Chunk Distribution*

Collection Chunks	On Shard
movies.drama	Shard0000
movies.action	Shard0001
movies.comedy	Shard0002

A **tag** describes the shard's property, which can be anything. Hence you might tag a shard as "slow" or "fast" or "rack space" or "west coast."

In the following example, you will tag the shards as belonging to each of the collection:

```
mongos> sh.addShardTag("shard0000", "dramas")
mongos> sh.addShardTag("shard0001", "actions")
mongos> sh.addShardTag("shard0002", "comedies")
mongos>
```

This signifies the following:

- Put the chunks tagged “dramas” on shard0000.
- Put the chunks tagged “actions” on shard0001.
- And put the chunks tagged “comedies” on shard0002.

Next, you will create rules to tag the collections chunk accordingly.
Rule 1: All chunks created in the `movies.drama` collection will be tagged as “dramas:”

```
mongos> sh.addTagRange("movies.drama", {originality:MinKey}, {originality:MaxKey}, "dramas")
mongos>
```

The rule uses MinKey, which means negative infinity, and MaxKey, which means positive infinity. Hence the above rule means mark all of the chunks of the collection `movies.drama` with the tag “dramas.”
Similar to this you will make rules for the other two collections.
Rule 2: All chunks created in the `movies.action` collection will be tagged as “actions.”

```
mongos> sh.addTagRange("movies.action", {distribution:MinKey}, {distribution:MaxKey}, "actions")
mongos>
```

Rule 3: All chunks created in the `movies.comedy` collection will be tagged as “comedies.”

```
mongos> sh.addTagRange("movies.comedy", {collection:MinKey}, {collection:MaxKey}, "comedies")
mongos>
```

You need to wait for the cluster to rebalance so that the chunks are distributed based on the tags and rules defined above. As mentioned, the chunk distribution is an automatic process, so after some time the chunks will automatically be redistributed to implement the changes you have made.
Next, issue `chunks.find` to vet the chunks organization:

```
mongos> use config
switched to db config
mongos> db.chunks.find({ns:"movies.drama"}, {shard:1, _id:0}).sort({shard:1})
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
mongos> db.chunks.find({ns:"movies.action"}, {shard:1, _id:0}).sort({shard:1})
```

```
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
mongos> db.chunks.find({ns:"movies.comedy"}, {shard:1, _id:0}).sort({shard:1})
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
mongos>
```

Thus the collection chunks have been redistributed based on the tags and rules defined (Figure 7-23).

Figure 7-23. *Distribution with tagging*

Scaling with Tagging

Next, you will look at how to scale with tagging. Let's change the scenario. Let's assume the collection `movies.action` needs two servers for its data. Since you have only three shards, this means the other two collection's data need to be moved to one shard.

In this scenario, you will change the tagging of the shards. You will add the tag "comedies" to Shard0 and remove the tag from Shard2, and further add the tag "actions" to Shard2.

This means that the chunks tagged "comedies" will be moved to Shard0 and chunks tagged "actions" will be spread to Shard2.

You first move the collection `movies.comedy` chunk to Shard0 and remove the same from Shard2:

```
mongos> sh.addShardTag("shard0000","comedies")
mongos> sh.removeShardTag("shard0002","comedies")
```

Next, you add the tag "actions" to Shard2, so that `movies.action` chunks are spread across Shard2 also:

```
mongos> sh.addShardTag("shard0002","actions")
```

Re-issuing the find command after some time will show the following results:

```
mongos> db.chunks.find({ns:"movies.drama"}, {shard:1, _id:0}).sort({shard:1})
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
mongos> db.chunks.find({ns:"movies.action"}, {shard:1, _id:0}).sort({shard:1})
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
mongos> db.chunks.find({ns:"movies.comedy"}, {shard:1, _id:0}).sort({shard:1})
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
```

```
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
mongos>
```

The chunks have been redistributed reflecting the changes made (Figure 7-24).

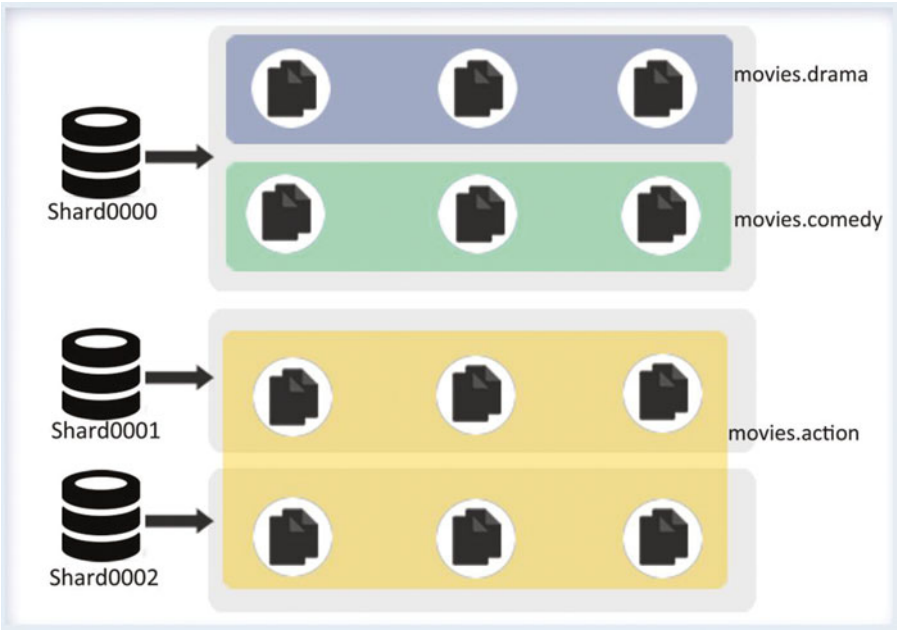


Figure 7-24. *Tagging with scaling*

Multiple Tags

You can have multiple tags associated with the shards. Let's add two different tags to the shards.

Say you want to distribute the writes based on the disk. You have one shard that has a spinning disk and the other has a SSD (solid state drive). You want to redirect 50% of the writes to the shard with SSD and the remaining to the one with the spinning disk.

First, tag the shards based on these properties:

```
mongos> sh.addShardTag("shard0001", "spinning")
mongos> sh.addShardTag("shard0002", "ssd")
mongos>
```

Let's further assume you have a `distribution` field of the `movies.action` collection that you will be using as the shard key. The `distribution` field value is between 0 and 1. Next, you want to say, "If distribution < .5, send this to the spinning disk. If distribution >= .5, send to the SSD." So you define the rules as follows:

```
mongos>sh.addTagRange("movies.action", {distribution:MinKey} ,{distribution:.5} ,"spinning")
mongos>sh.addTagRange("movies.action" ,{distribution:.5} ,{distribution:MaxKey},"ssd")
mongos>
```

Now documents with distribution < .5 will be written to the spinning shard and the others will be written to the SSD disk shard.

With tagging you can control the type of load that each newly added server will get.

Points to Remember When Importing Data in a ShardedEnvironment

Here are some points to keep in mind when importing data.

Pre-Splitting of the Data

Instead of leaving the choice of chunks creation with MongoDB, you can tell MongoDB how to do so using the following command:

```
db.runCommand( { split : "practicalmongodb.mycollection" , middle : { shardkey : value } } );
```

Post this you can also let MongoDB know which chunks goes to which node.

For all this you will need knowledge of the data you will be imported to the database. And this also depends on the use case you are aiming to solve and how the data is being read by your application. When deciding where to place the chunk, keep things like data locality in mind.

Deciding on the Chunk Size

You need to keep the following points in mind when deciding on the chunk size:

1. If the size is too small, the data will be distributed evenly but it will end up having more frequent migrations, which will be an expensive operation at the mongos layer.
2. If the size is large, it will lead to less migration, reducing the expense at the mongos layer, but you will end up with uneven data distribution.

Choosing a Good Shard Key

It's very essential to pick a good shard key for good distribution of data among nodes of the shard cluster.

Monitoring for Sharding

In addition to the normal monitoring and analysis that is done for other MongoDB instances, the sharding cluster requires an additional monitoring to ensure that all its operations are functioning appropriately and the data is distributed effectively among the nodes. In this section, you will see what monitoring you should do for the proper functioning of the sharding cluster.

Monitoring the Config Servers

The config server, as you know by now, stores the metadata of the sharded cluster. The mongos caches the data and routes the request to the respective shards. If the config server goes down but there's a running mongos instance, there's no immediate impact on the shard cluster and it will remain available for a while. However, you won't be able to perform operations like chunk migration or restart a new mongos. In the long run, the unavailability of the config server can severely impact the availability of the cluster. To ensure that the cluster remains balanced and available, you should monitor the config servers.

Monitoring the Shard Status Balancing and Chunk Distribution

For a most effective sharded cluster deployment, it's required that the chunks be distributed evenly among the shards. As you know by now, this is done automatically by MongoDB using a background process. You need to monitor the shard status to ensure that the process is working effectively. For this, you can use the `db.printShardingStatus()` or `sh.status()` command in the mongos mongo shell to ensure that the process is working effectively.

Monitoring the Lock Status

In almost all cases the balancer releases its locks automatically after completing its process, but you need to check the lock status of the database in order to ensure there's no long lasting lock because this can block future balancing, which will affect the availability of the cluster. Issue the following from mongos mongo to check the lock status:

```
use config
db.locks.find()
```

Production Cluster Architecture

In this section, you will look at the production cluster architecture. In order to understand it, let's consider a very generic use case of a social networking application where the user can create a circle of friends and can share their comments or pictures across the group. The user can also comment or like her friend's comments or pictures. The users are geographically distributed.

The application requirement is immediate availability across geographies of all the comments; data should be redundant so that the user's comments, posts and pictures are not lost; and it should be highly available. So the application's production cluster should have the following components:

1. At least two mongos instance, but you can have more as per need.
2. Three config servers, each on a separate system.
3. Two or more replica sets serving as shards. The replica sets are distributed across geographies with read concern set to nearest. See Figure 7-25.

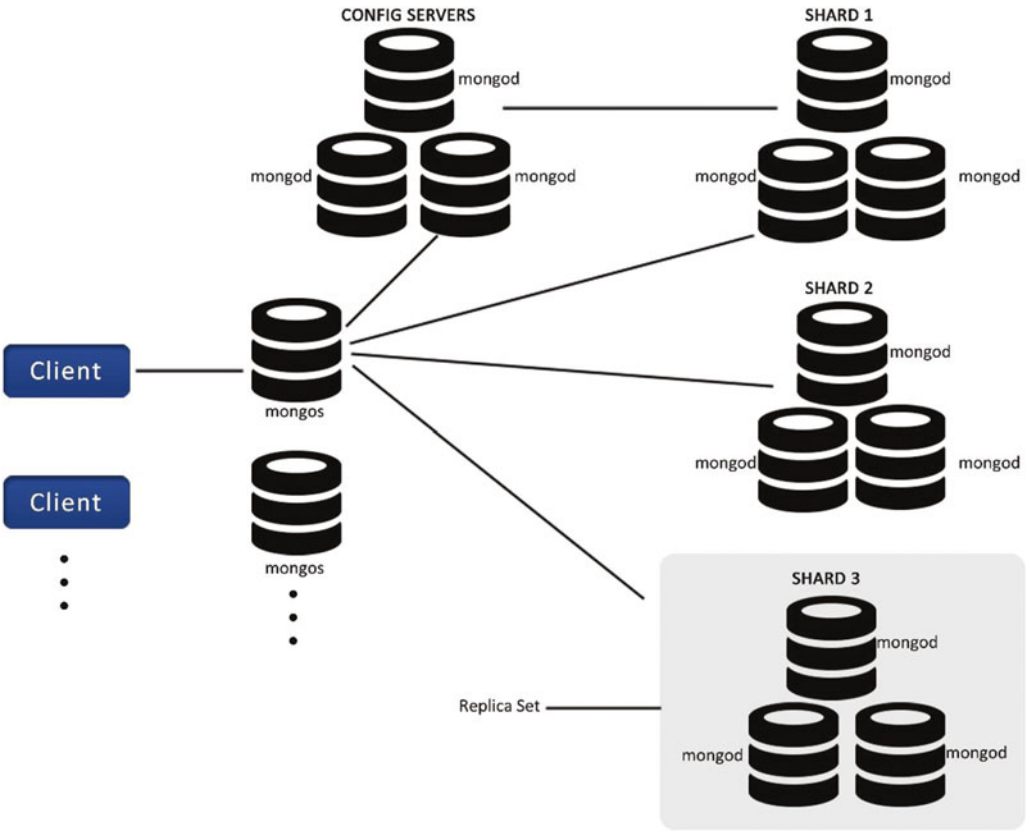


Figure 7-25. *Production cluster architecture*

Now let's look at the possible failure scenarios in MongoDB production deployment and its impact on the environment.

Scenario 1

Mongos become unavailable: The application server where mongos has gone down will not be able to communicate with the cluster but it will not lead to any data loss since the mongos don't maintain any data of its own. The mongos can restart, and while restarting, it can sync up with the config servers to cache the cluster metadata, and the application can normally start its operations (Figure 7-26).

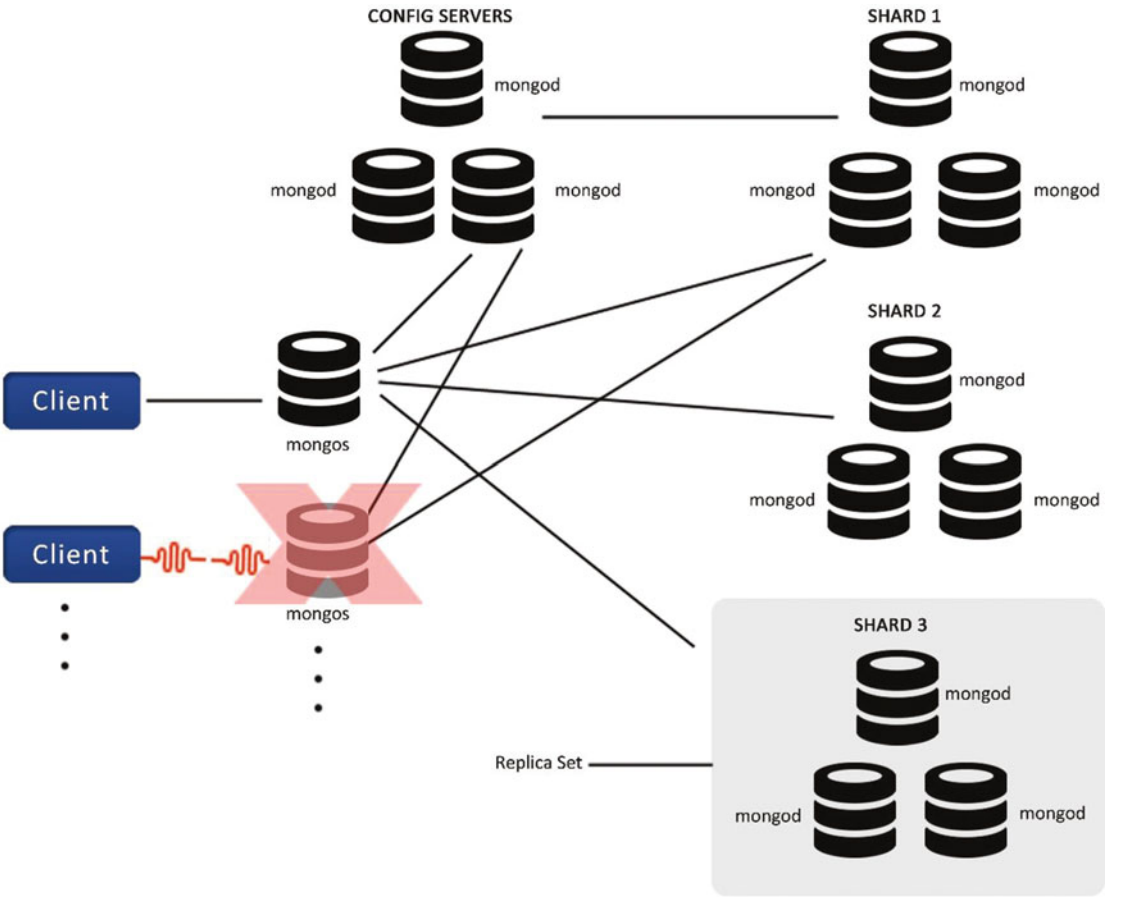


Figure 7-26. *Mongos become unavailable*

Scenario 2

One of the mongod of the replica set becomes unavailable in a shard: Since you used replica sets to provide high availability, there is no data loss. If a primary node is down, a new primary is chosen, whereas if it's a secondary node, then it is disconnected and the functioning continues normally (Figure 7-27).

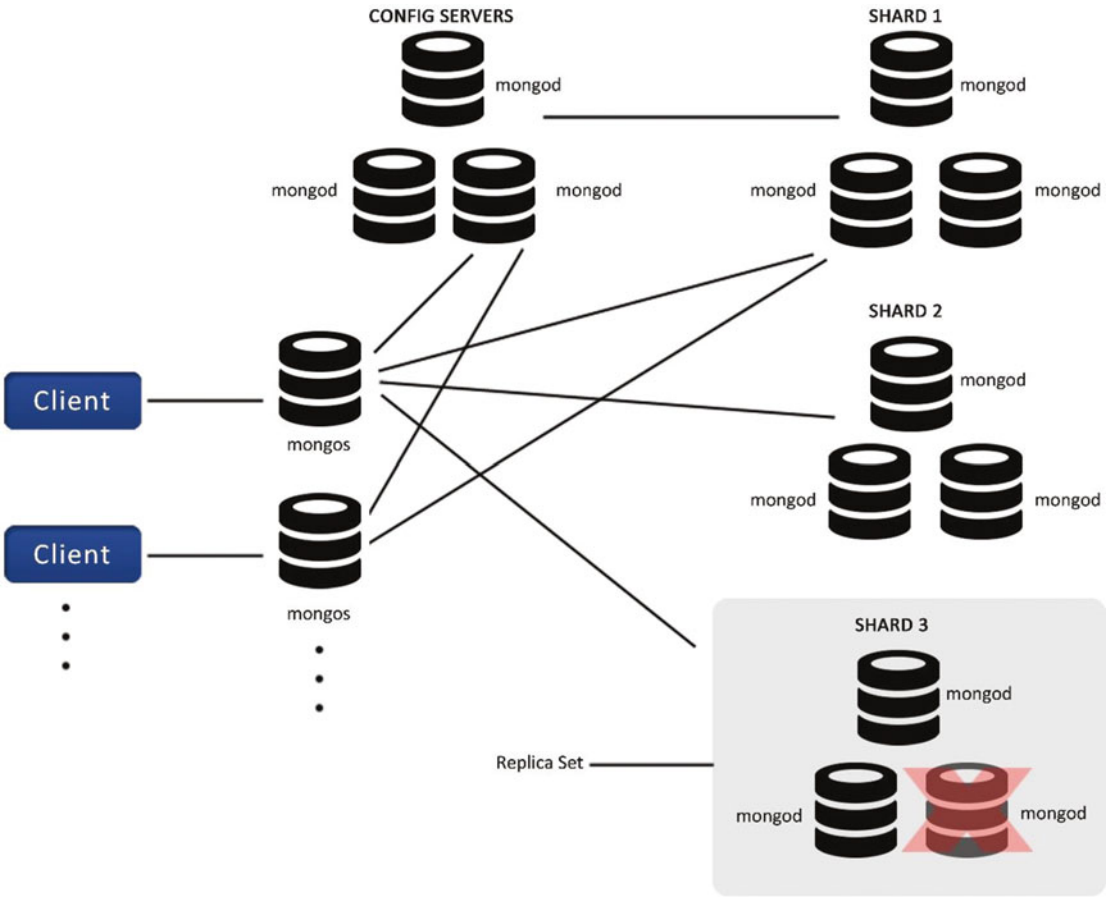


Figure 7-27. *One of the mongod of replica set is unavailable*

The only difference is that the duplication of the data is reduced, making the system little weak, so you should in parallel check if the mongod is recoverable. If it is, it should be recovered and restarted whereas if it's non-recoverable, you need to create a new replica set and replace it as soon as possible.

Scenario 3

If one of the shard becomes unavailable: In this scenario, the data on the shard will be unavailable, but the other shards will be available, so it won't stop the application. The application can continue with its read/write operations; however, the partial results must be dealt with within the application. In parallel, the shard should attempt to recover as soon as possible (Figure 7-28).

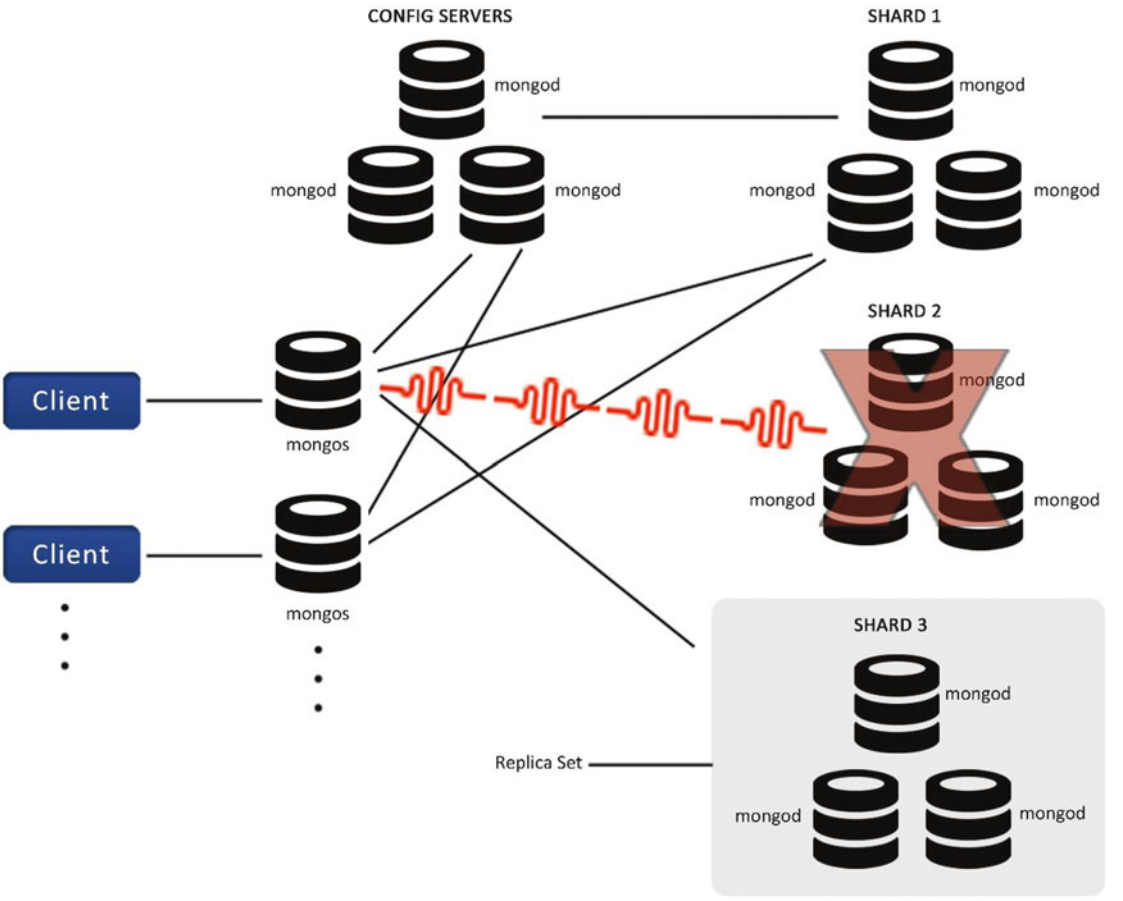


Figure 7-28. *Shard unavailable*

Scenario 4

Only one config server is available out of three: In this scenario, although the cluster will become read-only, it will not serve any operations that might lead to changes in the cluster structure, thereby leading to a change of metadata such as chunk migration or chunk splitting. The config servers should be replaced ASAP because if all config servers become unavailable, this will lead to an inoperable cluster (Figure 7-29).

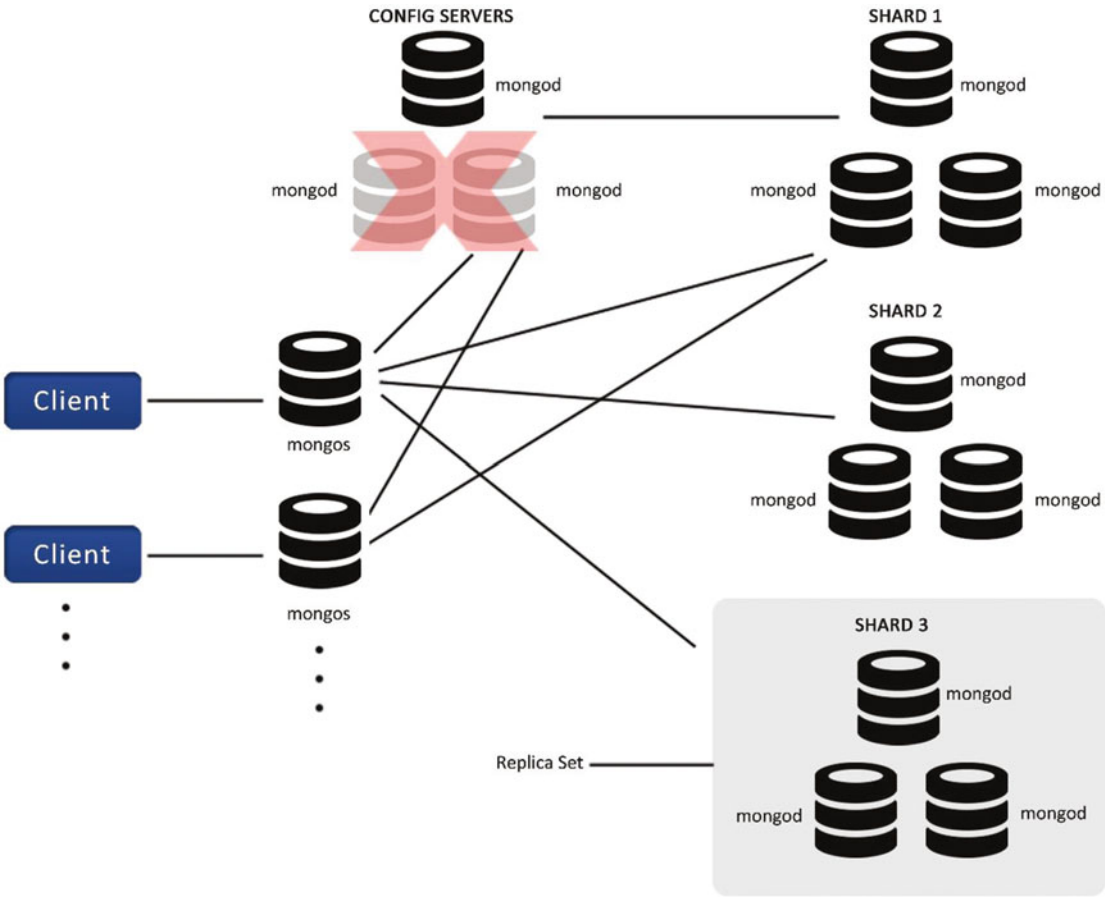


Figure 7-29. *Only one config server available*

Summary

In this chapter you covered the core processes and tools, standalone deployment, sharding concepts, replication concepts, and production deployment. You also looked at how HA can be achieved.

In the following chapter, you will see how data is stored under the hood, how writes happens using journaling, what is GridFS used for, and the different types of indexes available in MongoDB.